A SEMANTIC FOUNDATION FOR SOUND GRADUAL TYPING

MAX STEWART NEW

2020

For Sam.

ABSTRACT

Gradually typed programming languages provide a way forward in the debate between static and dynamic typing. In a gradual language, statically typed and dynamically typed programs can intermingle, and dynamically typed scripts can be gradually migrated to a statically typed style. In a *sound* gradually typed language, static type information is just as reliable as in a static language, establishing correctness of type-based refactoring and optimization. To ensure this in the presence of dynamic typing, runtime type casts are inserted automatically at the boundary between static and dynamic code. However the design of these languages is somewhat ad hoc, with little guidance on how to ensure that static reasoning principles are valid.

In my dissertation, I present a semantic framework for design and metatheoretic analysis of gradually typed languages based on the theory of embedding-projection pairs. I show that this semantics enables proofs of the fundamental soundness theorems of gradual typing, and that it is robust, applying it to different evaluation orders and programming features.

ACKNOWLEDGMENTS

First, I thank my advisor Amal Ahmed, who gave me a great deal of freedom in my PhD to pursue the ideas in this work with little evidence they would yield much of interest at the time. No one has influenced my techniques or my tastes in research as much as you. I also thank the members of my committee: Matthias Felleisen, Mitchell Wand, Daniel R. Licata, Ronald Garcia and Peter Thiemann, who had a larger task than most. In particular, I thank Matthias Felleisen who provided many interesting discussions on gradual typing that led directly to some of the results in this document. Also Dan Licata, who in many ways was a second advisor to me.

I want to thank everyone who I knew in the PRL lab throughout my years at Northeastern. Especially to William J. Bowman, PhD., who was a true mentor to me. I also need to thank Gabriel Scherer for both ruining me forever by explaining focusing to me and also supporting my strong desire to apply type theory and category theory to my research. And I especially thank the Happy Hour crew throughout the years: BLerner, Brian, Billy, Vincent, Tony, Asumu, Jr. Slepak, Andrew, Lief, Greenman, Hyeyoung, Aaron.

I want to thank my parents have supported me unconditionally my entire life and I am eternally grateful for that. And finally, I want to thank my partner Sam, who has supported me through this process almost as long as we've known each other, through every high and low. I certainly could not have gotten through this without you.

SUPPORT

This work was funded by the National Science Foundation under grants CCF-1422133, CCF-1453796, CCF-1816837, and CCF-1910522. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding agencies.

CONTENTS

LIST OF FIGURES

Part I

GRADUAL TYPING AND EMBEDDING-PROJECTION PAIRS

1

STATICALLY, DYNAMICALLY AND GRADUALLY TYPED PROGRAMMING LANGUAGES

The task of computer programming can be a very frustrating experience. Never before the rise of computers have humans had access to such a general-purpose form of automation. But also never before in human history have we been so painfully confronted with the need for *precision* in expressing our intentions or with such frequent reminders of the difficulty that we have in expressing ourselves correctly than we do when programming. The fact that semi-technical words like "bug", "crash", "freeze" have entered into the common language is a consequence of not only the ubiquity of software systems in our everyday lives, but also the difficulty that we still have in implementing software that functions as we desire it to.

And so with the rise in software there has been a corresponding rise in techniques for reducing these programming errors: informal specifications, model checking, unit testing, integration testing, property testing, static analysis, dynamic analysis, type systems, software contracts, fuzzing, and surely many others. All of these systems rely on one fundamental idea for reducing errors in programming: *do* repeat yourself. That is, express your intentions about the software in at least two ways, and then check for consistency between them. Often, this means giving the code and another, more coarse-grained specification, written in a more declarative style. For instance, with static type checking, we write the code that specifies the program's behavior precisely, but then we also give static type annotations that typically specify the behavior at a much more coarse-grained level. A static type checker checks the code for consistency with the type annotations. If they match, the code can be run, but if they do not match we have some kind of inconsistency between the two and the programmer then must determine which of the code or the type annotations (or neither) correctly represented their intentions for the program. Due to fundamental computational limitations static type checkers must be inherently conservative, rejecting programs which do actually satisfy the specification, but in a way that is too complex for the type checker to understand. As we will expand on soon, languages with non-trivial type checkers are called *statically typed*, while languages for which there are few to no type distinctions are called *dynamically typed*.

Here we mean no type distinctions are enforced by the language. A programmer always does have some type distinctions in mind while programming.

Another system for detecting programming errors is that of software contracts. Contracts are similar to types in that they are given alongside code and typically provide a coarse-grained specification for the program's behavior. However contracts are typically checked *while*

running the program, dynamically monitoring the execution for inconsistency with the contract. Delaying the checking to runtime means catching programming errors later, but also means that contracts need never have false positives: if a contract error is raised at runtime it is a true inconsistency between the contract and the program's behavior. Additionally, while static type systems are typically highly integrated into the language itself, contracts are a *post hoc* mechanism, and so a contract can be imposed on code that was not originally written with the contract in mind.

This dissertation is concerned with the semantics and design of *gradually typed* programming languages. These languages, studied as their own class since the mid-2000s [69, 81], are designed to support both dynamically typed and statically typed sublanguages, while facilitating the migration from a more dynamically typed style to the more statically typed style. That is, they allow for the gradual introduction of more precise specifications for code in the form of typing. So a program might be written in the dynamically typed sublanguage at first, but then one module might be migrated to static typing and type checked, then others. One of the primary applications is to add a static type system to an existing dynamically typed language, to give programmers using this language some of the benefits of static type checking.

To ensure that types remain reliable even while mixing statically typed and dynamically typed components, runtime-checking similar to software contracts is used at the boundaries between dynamically typed and statically typed styles. Much of the semantic difficulty of gradual typing lies in ensuring that these runtime checks correctly ensure the same kind of reasoning that the static types provide.

Before diving into more details of gradually typed languages and the challenges in designing them, we first review the basics of dynamic and static typing. Note that these terms (static and dynamic typing) are not universally agreed-upon, so we will try to provide a coherent, if not completely precise, working definition for the purposes of this dissertation.

1.1 STATIC AND DYNAMIC TYPING

Statically typed programming languages provide a formally verified mechanism for expressing interfaces between different program components. These interfaces are called *types* and when a program is built up using the languages' constructs, the type checker verifies that the components fit together properly.

This intrinsic view of typing as corresponding to "sorts" or "boundaries of composition" is well formalized by approaches to multi-sorted algebras and category theory.

A programming language is called dynamically typed if it lacks a mechanism for expressing some form of static type *distinction*.

So despite being defined by their *lack* of a feature, dynamically typed languages are quite popular, and many programmers espouse

a preference for dynamically typed languages. Why? First, it may be because the static types are burdensome to write down. This criticism holds especially true of many older statically typed that lack much means of type inference, but also can hold for languages with very expressive type systems that can express complex invariants that are difficult to formally check. Second, the static type system may be ill-adapted to formalizing the interfaces the programmer has in mind because it is not expressive enough to precisely describe the invariants in a program.

For these reasons a programmer may prefer a language that does little or no static type checking because it is easier to write a program that will be accepted by the compiler or interpreter and the programmer can get to the task of *running* the program and evaluating its behavior explicitly by interacting with it or running a test suite.

On the other hand, there are clear downsides to having a single "universal" interface. Usually when an operation is defined it has an intended domain of definition. For instance, a programmer is implementing a procedure that overlays one image on top of another, they are probably focusing on what the procedure does when its inputs are actually values that represent images. However, in most dynamically typed languages, there are many other kinds of values: bytestrings, unicode strings, numbers, structures of other kinds. In a dynamically typed language, every operation must have a defined behavior. For every primitive operation, the language must specify some behavior even on "unintended" inputs. One solution, the one most commonly associated to dynamic typing, is for the program to enter an "error state", by raising an exception or terminating the program and yielding control to the operating system. However, operations might instead return a sentinel value that is intended to represent an error, or they might simply return a value that might be valid, putting the onus on the user of the procedure to ensure that the inputs are valid.

Some languages such as C and C++ have "undefined behavior" meaning the implementation can do absolutely anything, but most languages intend for there to be a well-defined semantics for any program accepted by the compiler/interpreter.

This introduces a major advantage of static typing over dynamic typing: when the programmer's thinking is in sync with the type system, it simplifies reasoning about the program. If the types accurately reflect the intended inputs and outputs of a procedure, then the programmer only has to think about those inputs and outputs. In the case that the dynamically typed language would error with invalid inputs, this can be understood as eliminating the possibility of errors, which is a common perspective on the advantage of static typing. This is typically codified as a global property of a static type system as follows. First, a dynamically typed language is presented which raises errors or "gets stuck" and does not further evaluate when invalid inputs are passed to an operation. Then a static language is presented with nearly the same surface syntax except for some annotations related to typing information. Finally, it is proven that the programs that pass the type checker never produce any errors or get

stuck. However this global approach is at odds with the *gradual* part of gradual typing, where static types are only added a little at a time.

1.2 DYNAMICALLY TYPED LANGUAGES

Dynamically typed programming languages are characterized by their *lack* of static type distinctions, i.e., operations such as function application or arithmetic operations are considered valid on any syntactically well-formed subterms, rather than requiring the subterms to satisfy some particular types. From a static typing perspective, we can think of dynamically typed programming languages as those only admitting one single type, which need not be named since there are no other things that it need be distinguished from. This perspective on dynamically typed languages as being a special case of statically typed languages goes back to the earliest work on programming language semantics [67]. This viewpoint is summarized with the idea (explained, and attributed to Dana Scott in [36]) that *un*typed languages are really *uni*typed languages, i.e. they have only one type, the universal type of all well-formed terms, and so there are no *distinctions* between types of term.

This does not mean there is no "static checking" of dynamically typed programs: for instance, many dynamically typed languages have some static checking to ensure that all variables are bound before the program is run.

Another common feature that is however not universal is that many built-in operations raise *errors* when given "invalid" inputs. While generally dynamically typed languages feature "more" of these operations, most statically typed languages feature them as well: division for example is typically defined for a type that includes 0 and an error is raised at runtime if 0 is passed as the denominator. Additionally, dynamically typed languages don't necessarily raise errors when invalid inputs are passed to built-in functions. For instance, JavaScript returns a special 'undefined' value in many cases, and so moving from dynamic to static typing might not inherently prevent runtime errors.

1.2.1 *Migrating from Dynamic to Static Typing*

When faced with the choice of whether to use a statically typed or dynamically typed programming language for a project, a programmer must weigh the perceived benefits of the two styles. Crucially, we must consider the *context* of the programmer's decision. Maybe the program must be written urgently and an initial version of the program might

In fact in well-behaved gradually typed languages, adding types to some *of a program results in potentially* more errors *because a stricter type discipline is imposed at runtime.*

1.3 GRADUAL TYPING

Gradually typed languages are those that accommodate some kind of type distinctions while also allowing for a mode of programming where there are *no* type distinctions, and, most importantly, allowing for programs written in the two styles to interoperate. Very often, such a language is designed by taking an existing dynamically typed

language and adding a static typing discipline to it, with the intent to support a simple migration path from the dynamic to static discipline. Importantly, the language is designed to allow for *gradual* migration, that is the programmer can migrate one module (or function or expression) at a time while still being able to typecheck and run their code. The motivation for this gradual migration is that this is easier and less error-prone than rewriting a system from one language to another that has very different syntax and semantics. In this way, a programmer using the language can gradually accrue the benefits of static typing, i.e. the reliable static reasoning principles!

The idea of adding a static type system to a dynamic language for the purposes of improved compilation has precedent in Common Lisp, though the soundness of the types is not enforced. The modern idea of a gradually typed language was codified in two papers published at roughly the same time. Tobin-Hochstadt and Felleisen [82] developed Typed Scheme (now Typed Racket), a type system for the dynamically-typed language PLT Scheme (now Racket), with the ability for statically typed and dynamically typed modules to interoperate, with contracts inserted to ensure the soundness of typing. Siek and Taha [69] introduced the Gradually Typed Lambda Calculus, a typed language with a special "dynamic type" that is treated specially by the type checker in a way that allows for a lax static typing discipline like that of a dynamically typed language. The semantics of these two styles is the focus of our work, and we introduce their formal syntax and operational semantics in Chapter 2.

There is a third, increasingly popular category of languages, which we refer to as "optionally typed", that falls outside the purview of this dissertation. For these languages, like TypeScript [10, 37] and Hack [86], the static type system is used as a kind of simple static analysis in that it is meant to find bugs, but not meant to provide reliable reasoning principles for programmers. The runtime semantics of these languages is essentially the same as a dynamically typed language. The tradeoff for these languages is that while the types do not provide reliable reasoning principles, there is no "interoperability overhead" between statically and dynamically typed components. Since their semantics is essentially the same as dynamic typing, they do not exhibit the same issues as the "sound" gradually typed languages in the style of [69, 82] and so are out of scope for this dissertation.

1.4 REASONING ABOUT PROGRAMS

Programming Languages are designed to aid in program reasoning. One of the key tools of the programming language designer is the study of relational properties of programs. Why relational properties? The key distinction between the theory of programming languages and the theory of computation is that the theory of computation is

chiefly concerned with properties of single programs: Can a program solve this problem? How efficiently can a program solve this problem? If we add a language feature such as angelic non-determinism can a single program solve the same problem faster? The theory of programming languages, on the other hand, is concerned with such questions as "What will happen if I change a certain subexpression of a program?" and "Can this data representation be replaced with another?" and "Can this assertion be removed without affecting the programs semantics?"

We can think of all these questions in one unified view: a main focus theory of programming languages is the study of the *changes* in programs over time. More specifically, we focus on how *syntactic changes* affect *semantics*, and studying syntactic principles that codify these reasoning abilities. We can think of the life cycle of a program as a kind of discrete dynamical system, with the programmer making changes over time. Most of the semantic properties we will discuss and prove in this dissertation will be of the form "X syntactic change" results in "Y semantic change". The three main semantic properties that we consider in this dissertation: graduality, $\beta\eta$ equivalence, and parametricity, can all be understood within this framework. First, *graduality* tells the programmer that when they change the types in their program to be more precise, but don't otherwise change the code, then they have only increased the precision of runtime enforcement, and so the program will either behave the same, or raise a dynamic error it did not before. This simplifies reasoning about the correctness when adding types because the programmer can add more precise types without affecting partial program correctness. Next, *$\beta\eta$ equivalence* justifies many different program transformations that do not change the extensional behavior of programs and justify things like inlining, simplifications, etc. This supports refactoring by the programmer in addition to improved compilation. Finally, *parametricity* gives sufficient conditions for reasoning about changing the underlying representation of abstract data types, allowing a programmer to start out with an inefficient implementation and swap out an equivalent more efficient implementation later.

1.5 THESIS

This dissertation presents a semantic foundation for gradually typed languages. That is, a theory of gradual typing runtime semantics that aids in designing and analyzing gradually typed languages. The kernel of the theory is that the runtime type enforcement of gradually typed languages can usefully be structured in terms of the mathematical concept of *embedding-projection pairs*. My thesis is that

> The theory of embedding-projection pairs provides a common semantic framework for the design and metatheory of sound gradually typed languages.

The basic idea (expanded in Chapter 3) is to give a semantics where the meaning of a *gradual* type consists of two components: the *logical* type of inhabitants paired with a pair of functions that say how to *cast* to and from the "universal" dynamic type. One function *embeds* inhabitants of the type in the universal type, defining the "dynamically typed interface" to the values of the type. The other function *projects* inhabitants of the dynamic type down into the logical type. This projection is a *partial* operation, it errors when the dynamic value does not fit into the logical type. These two functions, the embedding and the projection, both define the dynamic interface to the type, and so then we impose some additional conditions that say that they agree. First, if we embed a value of the type into the universal type, the projection function should agree that this is a well-formed value, and so projecting the value back to the logical type should get us a value that is equivalent to the original in observable behavior. Next, if we project a value down to the logical type, it will have the type enforced, so it may error now, or later on if it is a higher order type. So if we embed this back into the dynamic type, we should get something that is like the original value, but has some "enforcement" now built into it. We specify this by saying that the value should "error more" than the original.

The goal of the dissertation is to show that these notions are a natural way to structure the semantics of gradually typed languages that satisfy the graduality property and provide relational reasoning principles. We will show that the graduality property relies critically on the defining properties of embedding-projection pairs. We also show that relational reasoning principles for the surface language follow from the corresponding principles for of the target language of the elaboration.

1.6 CONTRIBUTIONS AND STRUCTURE OF THESIS

The dissertation is structured in three parts. In Part I I show how the embedding-projection pair semantics can be used to prove the desired properties of a gradually typed language. In Chapter 2, I show two representative formal syntaxes of gradually typed languages, and the desired reasoning properties for their operational semantics. Then in Chapter 3, I introduce the embedding-projection pair semantics and prove our desired graduality and relational reasoning properties.

In Part II, I explore to what extent we can derive what the semantics of gradual typing should be from these desired properties. In Chapter 5, I define Gradual Type Theory (GTT), which axiomatizes the graduality and equational reasoning principles we desire of gradual

typing, based on the ep-pair properties. We then show that nearly all of the operational semantics of gradual typing for simple types is fully determined by this axiomatization. In Chapter 7, I show the consistency of this axiomatic theory by giving semantic models, and in Chapter 6 I show how the theory is general enough to derive gradual operational semantics for three different evaluation orders.

In Part III, I explore the problem of designing a gradually typed language that supports parametric polymorphism while satisfying the crucial parametricity theorem. In Chapter 9, I show that at least one interpretation of this problem is impossible to solve, in that enforcing parametricity is uncomputable. In Chapter 10 I explore previous work which satisfies parametricity but not graduality, and show how to modify the language so that the semantics is fully explainable by an embedding-projection pair translation, and both parametricity and graduality properties fall out easily.

Finally in Chapter 11 we discuss some of the existing limitations of our approach and some avenues for future work.

2

SYNTAX AND SEMANTICS OF GRADUALLY TYPED LANGUAGES AND CAST CALCULI

In this chapter, I present two different calculi whose syntaxes are representative of common approaches to the syntax of a gradually typed language, and how to equip them with an operational semantics by elaboration to a cast calculus. The first style I will call *multilanguage* gradual typing and is most similar to gradual languages like Typed Racket [82]. The second approach I will call *fine-grained* and is based on Siek and Taha's gradually typed lambda calculus[69]. Despite significant syntactic differences, the two styles are semantically quite similar, and work in later chapters will work primarily with *cast calculi*, which are core languages designed as a target of an elaboration from the surface syntax. We will give semantics to both the multilanguage and fine-grained calculi by elaboration to a single cast calculus. Finally, we discuss some syntactic reasoning principles and how they manifest in the two different language styles.

2.0.1 *Multilanguage Gradual Typing*

The philosophy behind multilanguage gradual typing is that the gradual language G consists of two interoperating languages: D the dynamically typed language and S the statically typed variant of the language D. To facilitate migration, S should be as similar in syntax to D as is possible while accommodating the need for type annotations. Then the two languages interact by a sort of high-level FFI where the typed code assigns types when importing from the dynamic language. This interface is made at the level of the module system, where statically typed modules can import from dynamic modules and vice-versa, but in order to have a simple expression-based calculus, we'll adopt the "multi-language" approach following [49] which has been used for gradual typing calculi before[33, 88].

Figure 2.1 contains the syntax and typing of a simple gradual language using the multi-language approach. The two sublanguages are a simple dynamically typed language and a simply typed lambda calculus, with two "language boundaries": one to import dynamically typed values into statically typed code and vice-versa. Note that the context here consists of both typed variables, written "$x : A$" and dynamic variables written "x dynamic".

$$
\begin{array}{rrcl}
\text{types} & A, B & ::= & \texttt{Bool} \mid A \times B \mid A \to B \\
\text{static terms} & t, u & ::= & x \mid \text{ToType}^A t \mid t\ u \mid \lambda x : A.t \mid (t, u) \mid \pi_i t \\
& & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u \\
\text{dynamic terms} & t, u & ::= & x \mid \text{FromType}^A t \mid t\ u \mid \lambda x.t \mid (t, u) \mid \pi_i t \\
& & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u
\end{array}
$$

$$\frac{\Gamma \vdash^D t}{\Gamma \vdash^S \text{ToType}^A t : A} \qquad \frac{\Gamma \vdash^S t : A}{\Gamma \vdash^D \text{FromType}^A t}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash^S x : A} \qquad \frac{x\,\text{dynamic} \in \Gamma}{\Gamma \vdash^D x}$$

$$\frac{\Gamma \vdash^S t : A \to B \qquad \Gamma \vdash^S u : A}{\Gamma \vdash^S t\ u : B} \qquad \frac{\Gamma \vdash^D t \qquad \Gamma \vdash^D u}{\Gamma \vdash^D t\ u}$$

$$\frac{\Gamma, x : A \vdash^S t : B}{\Gamma \vdash^S \lambda x : A.t : A \to B} \qquad \frac{\Gamma, x\,\text{dynamic} \vdash^D t}{\Gamma \vdash^D \lambda x.t}$$

$$\frac{\forall i \in \{1,2\}.\Gamma \vdash^S t_i : A_i}{\Gamma \vdash^S (t_1, t_2) : A_1 \times A_2} \qquad \frac{\forall i \in \{1,2\}.\Gamma \vdash^D t_i}{\Gamma \vdash^D (t_1, t_2)}$$

$$\frac{\Gamma \vdash^S t : A_1 \times A_2}{\Gamma \vdash^S \pi_i t : A_i} \qquad \frac{\Gamma \vdash^D t}{\Gamma \vdash^D \pi_i t}$$

$$\Gamma \vdash^S \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash^S \texttt{false} : \texttt{Bool} \qquad \Gamma \vdash^D \texttt{true} \qquad \Gamma \vdash^D \texttt{false}$$

$$\frac{\Gamma \vdash^S t : \texttt{Bool} \qquad \Gamma \vdash^S u_t : A \qquad \Gamma \vdash^S u_f : A}{\Gamma \vdash^S \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f : A} \qquad \frac{\Gamma \vdash^D t \qquad \Gamma \vdash^D u_t \qquad \Gamma \vdash^D u_f}{\Gamma \vdash^D \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f}$$

Figure 2.1: Multi-language-Style Gradual Typing

2.0.2 *Fine-Grained Gradual Typing*

Next, in Figure 2.2, we present the syntax of what I will call *fine-grained* gradual typing, which is a popular style in the gradual typing literature, and originates with [69]. The philosophy behind fine-grained gradual typing, outlined here: [75], is that the gradual language G is essentially a statically typed language with a special type ? . Then rather than dynamically typed values being *explicitly* imported into a static type, type checking is relaxed at the boundary of static and dynamic code: dynamically typed values are allowed to flow to static code and vice-versa with no syntactic annotation from the programmer using a type-checking strategy similar to but distinct from subtyping. A good intuition of how the type checker works is to think of the dynamic type ? as representing "uncertainty" about a value and the type checker is an analysis that checks for if it is *plausible* that the program type checks.

in general, gradually typed languages do not necessarily have a ? type. Both because some are language-based and do not support a ? type and others have some other aspect of the type system that is gradual, so there might be a dynamic effect type [66] or dynamic row type [29] or dynamic refinement [44]

The first rule is the rule for type annotations $t :: A$. This says we can annotate t with a type A as long as its original type B is "consistent", written $A \sim B$ and pronounced "A is consistent with B". This notion of consistency is the main way laxness of checking is formalized in the type checker: $A \sim B$ holds when A and B are the same type, except where subformulae on one side or the other are ?. If we think of ? as representing uncertainty, this says it is plausible that A and B are the same type. This intuition can be formalized as a Galois connection where gradual types represent sets of possible static types. See the *Abstracting Gradual Typing* framework for much more on this approach to the design of fine-grained gradual surface languages[29]. The variable rule, and the introduction rules λ, pairs, true and false are all the same as for static typing. The elimination rules however, build in laxness of checking. For the application rule, we say that an application $t\,u$ is well typed if the domain of the function is *consistent* with the type of the input, and the resulting type is the codomain of the function. Here dom and cod are partial functions on gradual types that extract the best approximation to the domain and codomain of a type: for function types, the actual domain and codomain, but for the dynamic type, we extract the dynamic type. If the partial function is not defined for a type, then the rule does not hold. This use of partial functions makes the type checking more succinct, but we could equivalently express the function rule as multiple rules: one where the term in function position has a function type and one where it has the dynamic type.

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A' \qquad A \sim A'}{\Gamma \vdash t\,u : B} \qquad \frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : A'}{\Gamma \vdash t\,u : ?}$$

The product projections follow a similar structure: projecting a field has the most precise type for that field, given by the partial function

on types π_i. Finally, following Garcia and Cimini [28], the if-statement checks that the scrutinee is plausibly a boolean and the type of the term as a whole is the gradual *meet* of the two continuations. Gradual meet is also defined as a partial function at the bottom of the figure, with the meet of ? with any other type A being A and otherwise checking that the two types are structurally the same. An argument for why this should be meet and not join is that it agrees with simply-typed λ calculus on terms that do not contain ?. If it were instead gradual join, we would have terms where the programmer never explicitly uses any dynamic types, but they are introduced, for example:

$$\frac{\Gamma \vdash t : \texttt{Bool} \qquad \Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \Gamma \vdash \lambda x : \texttt{Bool}.x}{\Gamma \vdash \texttt{if}\ t\ \texttt{then true else}\ \lambda x : \texttt{Bool}.x : ?}$$

would not type check in simply typed lambda calculus, but it would typecheck with a dynamic type here. This is undesirable since one goal of gradual language is to support a static sublanguage with strict type checking.

2.0.3 *Multi-language vs Fine-Grained Gradual Typing*

The two gradually typed surface calculi each have advantages over the other.

First, for migration purposes, the multi-language approach makes all interaction between languages *explicit*, and supports the syntax of the original dynamically typed language directly. On the other hand, in the fine-grained approach, dynamically typed code must be annotated with many casts to the dynamic type ?, since introduction forms all are typed with a more precise type. For instance, a dynamically typed application function $\lambda f.\lambda x.f\ x$ would in the fine-grained language be written as

$$(\lambda f : ?.(\lambda x : ?.f x) :: ?) :: ?$$

so all introduction forms would be annotated with ?, but elimination forms do not.

On the other hand, by supporting a dynamic type ?, the fine-grained calculus can support some intermediate states of typing that cannot be represented in our multi-language approach. For instance, in the fine-grained calculus, we can gradually migrate a function from being dynamically typed ? to a function with dynamic domain and codomain $? \to ?$ and then gradually migrate the domain and codomain to more precise types. On the other hand, in our multi-language calculus, you must immediately pick a completely "static" type such as $\texttt{Bool} \to \texttt{Bool}$ to migrate to.

Modifications can be made to both calculi to overcome these difficulties, adopting something of a "hybrid" approach. For instance, we can add a dynamic sublanguage to the fine-grained calculus with similar

$$\begin{array}{rcl} \text{types } A, B & ::= & ? \mid \texttt{Bool} \mid A \times B \mid A \to B \\ \text{terms } t, u & ::= & x \mid t :: A \mid t\ u \mid \lambda x : A.t \mid (t, u) \mid \pi_i t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u \end{array}$$

$$\frac{\Gamma \vdash t : B \qquad A \sim B}{\Gamma \vdash t :: A : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A' \qquad \mathrm{dom}(A) \sim A'}{\Gamma \vdash t\ u : \mathrm{cod}(A)} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}$$

$$\frac{\forall i \in \{1,2\}.\Gamma \vdash t_i : A_i}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \pi_i t : \pi_i A} \qquad \Gamma \vdash \texttt{true} : \texttt{Bool}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool}$$

$$\frac{\Gamma \vdash t : A \qquad A \sim \texttt{Bool} \qquad \Gamma \vdash u_t : A_t \qquad \Gamma \vdash u_f : A_f}{\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f : A_t \sqcap A_f}$$

$$? \sim A \qquad A \sim ? \qquad \texttt{Bool} \sim \texttt{Bool} \qquad \frac{A \sim A' \qquad B \sim B'}{A \to B \sim A' \to B'}$$

$$\frac{A_1 \sim A_1' \qquad A_2 \sim A_2'}{A_1 \times A_2 \sim A_1' \sim A_2'}$$

$$\mathrm{dom}(?) = ?$$

$$\mathrm{dom}(A \to B) = A$$

$$\mathrm{cod}(?) = ?$$

$$\mathrm{cod}(A \to B) = B$$

$$\pi_i ? = ?$$

$$\pi_i (A_1 \times A_2) = A_i$$

$$? \sqcap A = A$$

$$A \sqcap ? = A$$

$$\texttt{Bool} \sqcap \texttt{Bool} = \texttt{Bool}$$

$$(A_1 \to A_2) \sqcap (B_1 \to B_2) = (A_1 \sqcap B_1) \to (A_2 \sqcap B_2)$$

$$(A_1 \times A_2) \sqcap (B_1 \times B_2) = (A_1 \sqcap B_1) \times (A_2 \sqcap B_2)$$

Figure 2.2: Fine-grained Gradual Typing

multi-language boundaries. The fine-grained calculus would then be the "static" portion of this multi-language, still with lax type checking when the dynamic type is used. Similarly, we can allow for more fine-grained migration in the multi-language by adding something like the dynamic type ? to the static types of the language. For instance, Typed Racket includes an "Any" type that is the top of its subtyping hierarchy. Adding this "Any" type does not mean we need to also accept the approach to gradual type-checking of the fine-grained calculus, instead we can think of Any as an uninterpreted base type for the purposes of static checking, or the top of a subtyping hierarchy.

So the true fundamental difference between the multi-language approach and the fine-grained approach is not that the multi-language approach features a multi-language boundary, or that the fine-grained approach supports more fine-grained use of the dynamic type, it is that in the multi-language approach, *all* interaction between dynamic and static code is *explicitly* annotated by the programmer with a multi-language or module boundary, whereas in the fine-grained approach, the lax type checking allows for syntactically *implicit* boundaries between dynamic and static types, driven by the presence of the dynamic type ?.

The implicit insertion of casts into the fine-grained calculi makes it difficult to determine where the elaboration inserts casts by inspection of a program. This difficulty is mitigated by the *graduality* property which gives reasoning principles for the addition of more precise types to a program.

2.1 CAST CALCULUS

A common approach to the *operational semantics* of gradual typing is given by *cast calculi*. Cast calculi typically include a dynamic type, but have a strict static type system. They provide a calculus in which the behavior of the built-in casts are specified in the form of some explicit cast term $\langle A \Leftarrow B \rangle t$. These casts can then be given an operational semantics. Both multilanguage and fine-grained styles of gradual typing can be given semantics by elaboration into the same cast calculus, showing that they can be built on the same semantic foundation.

We present the syntax of our cast calculus in Figure 2.3. The first rule gives the typing for the *cast* form that gives the calculus its name. This allows us to cast from any type A to an arbitrary other type B. The remaining rules for variables, functions/application, pairs/projection, booleans/if are the standard rules of the simply typed lambda calculus.

Next, we present the operational semantics of our cast calculus in Figure 2.4. First, we define "tag types" G to be the basic type constructors of the language applied to ? if they take any arguments. These represent the tags on dynamically typed values at runtime.

$$
\begin{array}{rrcl}
\text{types} & A, B & ::= & ? \mid \texttt{Bool} \mid A \times B \mid A \to B \\
\text{terms} & t, u & ::= & x \mid \langle A \Leftarrow B \rangle t \mid t\ u \mid \lambda x : A.t \mid (t, u) \mid \pi_i t \\
& & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ u\ \texttt{else}\ u
\end{array}
$$

$$
\frac{\Gamma \vdash t : B}{\Gamma \vdash \langle A \Leftarrow B \rangle t : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}
$$

$$
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \qquad \frac{\forall i \in \{1, 2\}.\ \Gamma \vdash t_i : A_i}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i t : A_i}
$$

$$
\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad\qquad \Gamma \vdash \texttt{false} : \texttt{Bool}
$$

$$
\frac{\Gamma \vdash t : \texttt{Bool} \qquad \Gamma \vdash u_t : A \qquad \Gamma \vdash u_f : A}{\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f : A}
$$

Figure 2.3: Cast Calculus Syntax and Typing

Then, we define our values V and evaluation contexts E, encoding a call-by-value, left-to-right evaluation order. The top of the figure shows the reductions *not* involving casts. This includes the standard reductions for pairs, sums, and functions using the usual notion of substitution $t[\gamma]$, in addition to a reduction $E[\mathcal{U}] \mapsto \mathcal{U}$ to propagate a dynamic type error to the top level.

More importantly, the bottom of the figure shows the reductions of *casts*, specifying the dynamic type checking necessary for gradual typing. First (DynDyn), casting from dynamic to itself is the identity. For any type A that is not a tag type (checked by $\lfloor A \rfloor \neq A$) or the dynamic type, casting to the dynamic type first casts to its underlying tag type $\lfloor A \rfloor$ and then tags it at that type (TagUp). Similarly, casting down from the dynamic type first casts to the underlying tag type (TagDn). The next two rules are the primitive reductions for tags: if you project at the correct tag type, you get the underlying value out (TagMatch) and otherwise a dynamic type error is raised (TagMismatch). Similarly, the next rule (TagMismatch') says that if two types are incompatible in that they have distinct tag types and neither is dynamic, then the cast errors. The next three (Fun, Pair, Sum) are the standard "wrapping" implementations of contracts/casts [25], also familiar from subtyping. For the function cast $\langle A_1 \to B_1 \Leftarrow A_2 \to B_2 \rangle$, note that while the output type is the same direction $\langle B_1 \Leftarrow B_2 \rangle$, the input cast is flipped: $\langle A_2 \Leftarrow A_1 \rangle$.

$$
\begin{array}{lcl}
G & ::= & ? \to ? \mid ? \times ? \mid \texttt{Bool} \\
V & ::= & \langle ? \Leftarrow G \rangle V \mid \lambda x : A.t \mid (V_1, V_2) \mid \texttt{true} \mid \texttt{false} \\
E & ::= & [\cdot] \mid \langle A \Leftarrow B \rangle E \mid E\, t \mid V\, E \mid (E, t_2) \mid (V_1, E) \mid \texttt{if } E \texttt{ then } u_t \texttt{ else } u_f
\end{array}
$$

$$E[\mho] \mapsto \mho \qquad E[(\lambda x : A.t)\, V] \mapsto E[t[V/x]] \qquad E[\pi_i(V_1, V_2)] \mapsto E[V_i]$$

$$E[\texttt{if true then } u_t \texttt{ else } u_f] \mapsto E[u_t]$$

$$E[\texttt{if false then } u_t \texttt{ else } u_f] \mapsto E[u_f]$$

$$E[\langle ? \Leftarrow ? \rangle V] \mapsto E[V] \qquad \frac{A \neq ? \qquad \lfloor A \rfloor \neq A}{E[\langle A \Leftarrow ? \rangle V] \mapsto E[\langle A \Leftarrow \lfloor A \rfloor \rangle \langle \lfloor A \rfloor \Leftarrow ? \rangle V]}$$

$$\frac{A \neq ? \qquad \lfloor A \rfloor \neq A}{E[\langle ? \Leftarrow A \rangle V] \mapsto E[\langle ? \Leftarrow \lfloor A \rfloor \rangle \langle \lfloor A \rfloor \Leftarrow A \rangle V]}$$

$$E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle V] \mapsto E[V] \qquad \frac{G \neq G'}{E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G' \rangle V] \mapsto \mho}$$

$$\frac{\lfloor A \rfloor \neq \lfloor B \rfloor}{E[\langle A \Leftarrow B \rangle V] \mapsto \mho}$$

$$E[\langle A \to B \Leftarrow A' \to B' \rangle V] \mapsto E[\lambda x : A. \langle B \Leftarrow B' \rangle (V\, (\langle A \Leftarrow A' \rangle x))]$$

$$E[\langle A_1 \times A_2 \Leftarrow A_1' \times A_2' \rangle (V_1, V_2)] \mapsto E[(\langle A_1 \Leftarrow A_1' \rangle V_1, \langle A_2 \Leftarrow A_2' \rangle V_2)]$$

Figure 2.4: Cast Calculus Operational Semantics

2.1.1 *Elaborating Surface Calculi*

Next, we define the elaboration for multilanguage syntax in Figure 2.5 and fine-grained syntax in Figure 2.6. The multilanguage elaboration can be divided into 3 pieces: elaboration of static terms, dynamic terms and boundaries. The static terms are elaborated directly into the corresponding constructor in the cast calculus. The boundaries are directly translated into casts involving the dynamic type. The dynamic terms are elaborated to use casts. Note that in general it is not necessarily the case that the dynamic language's semantics is the same as simply inserting casts into a lightly typed cast calculus program, because casts error on invalid inputs, but as mentioned in Chapter 1, dynamic languages like JavaScript return special undefined *values* instead. To accomodate this we would need to include some separate form in the cast calculus or explicitly describe the dynamic checking as a cast calculus program.

Next, we define the elaboration of the fine-grained syntax in Figure 2.6. First, annotations are translated directly into casts. Next, variables and introduction forms are directly translated to the corresponding construct of the cast calculus. Finally, elimination forms use an auxiliary function $G \wr t$ that inserts a cast on the scrutinee of the elimination form if the term is dynamically typed.

2.2 REASONING ABOUT GRADUALLY TYPED PROGRAMS

The elaborations of multilanguage and fine-grained gradual typing are markedly different, and the difference has an impact on the programmer using the language. The multi-language approach offers a very simple translation, essentially the dynamic code is a standard translation into typed code, while the typed code translation is a no-op other than the boundaries. The boundaries are all explicit and all runtime type checking is placed there. On the other hand, the fine-grained translation is complicated, with every elimination form possibly introducing dynamic type checking depending on the interaction between the types of different subterms. Then the dynamic type checking itself is given by the reductions of the cast calculus, which are themselves somewhat complicated. This naturally leads to the question of how *usable* this language is, i.e., how hard is it for a programmer to understand what their gradually-typed program's behavior will be?

In this section we consider program *reasoning principles* for our gradually typed calculi that provide some formal argument that while the specification of the language semantics is somewhat complex, there are some simple principles for understanding how the code behaves. These come in two classes. First, equational reasoning principles such as $\beta\eta$ equations tell us how to simplify programs and how type

$$
\begin{aligned}
.^{ml} &= . \\
(\Gamma, x : A)^{ml} &= \Gamma^{ml}, x : A^{ml} \\
(\Gamma, x\,\text{dynamic})^{ml} &= \Gamma^{ml}, x : ? \\
\\
(\text{ToType}^{A}\, t)^{S} &= \langle A \Leftarrow ? \rangle t^{D} \\
x^{S} &= x \\
(\lambda x : A.t)^{S} &= \lambda x : A.t^{S} \\
(t\, u)^{S} &= t^{S}\, u^{S} \\
(t_1, t_2)^{S} &= (t_1^{S}, t_2^{S}) \\
(\pi_i t)^{S} &= \pi_i t^{S} \\
\texttt{true}^{S} &= \texttt{true} \\
\texttt{false}^{S} &= \texttt{false} \\
(\texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f)^{S} &= \texttt{if}\ t^{S}\ \texttt{then}\ u_t^{S}\ \texttt{else}\ u_f^{S} \\
\\
(\text{FromType}^{A}\, t)^{D} &= \langle ? \Leftarrow A \rangle t^{S} \\
x^{D} &= x \\
(\lambda x.t)^{D} &= \langle ? \Leftarrow ? \to ? \rangle (\lambda x : ?.t^{D}) \\
(t\, u)^{D} &= (\langle ? \to ? \Leftarrow ? \rangle t^{D})\, u^{D} \\
(t_1, t_2)^{D} &= \langle ? \Leftarrow ? \times ? \rangle (t_1^{D}, t_2^{D}) \\
(\pi_i t)^{D} &= \pi_i \langle ? \times ? \Leftarrow ? \rangle t^{D} \\
\texttt{true}^{D} &= \langle ? \Leftarrow \texttt{Bool} \rangle \texttt{true} \\
\texttt{false}^{D} &= \langle ? \Leftarrow \texttt{Bool} \rangle \texttt{false} \\
(\texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f)^{D} &= \texttt{if}\ \langle \texttt{Bool} \Leftarrow ? \rangle t^{D}\ \texttt{then}\ u_t^{D}\ \texttt{else}\ u_f^{D}
\end{aligned}
$$

Figure 2.5: Elaboration of Multi-language Gradual Typing

$$
\begin{aligned}
(t :: B)^{fg} &= \langle B \Leftarrow A \rangle t^{fg} && \text{(where } t : A\text{)} \\
x^{fg} &= x \\
(\lambda x : A.t)^{fg} &= \lambda x : A.t^{fg} \\
(t\,u)^{fg} &= (? \to ? \wr t)\,(\langle \mathrm{dom}(A) \Leftarrow B \rangle u^{fg}) \\
& && \text{(where } t : A \text{ and } u : B\text{)} \\
(t_1, t_2)^{fg} &= (t_1{}^{fg}, t_2{}^{fg}) \\
(\pi_i t)^{fg} &= \pi_i (? \times ? \wr t) \\
\texttt{true}^{fg} &= \texttt{true} \\
\texttt{false}^{fg} &= \texttt{false} \\
(\texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f)^{fg} &= \texttt{if Bool} \wr t\ \texttt{then}\ \langle A \Leftarrow A_t \rangle u_t{}^{fg}\ \texttt{else}\ \langle A \Leftarrow A_f \rangle u_f{}^{fg} \\
& && \text{(where } u_t : A_t, u_f : A_f \text{ and } A = A_t \sqcap A_f\text{)} \\
G \wr t &= \langle G \Leftarrow ? \rangle t^{fg} && \text{(when } t : ?\text{)} \\
G \wr t &= t^{fg} && \text{(otherwise)}
\end{aligned}
$$

Figure 2.6: Elaboration of Fine-grained Gradual Typing

information informs program reasoning. These equational reasoning principles tell us what our type information is actually good for. Second, we have the inequational principle of *graduality*, which helps us reason about casts. The graduality principle ensures that, despite the fact that the semantics of fine-grained gradual typing is quite sensitive to the exact types that are present, the behavior of code that contains casts is easy to understand at the level of *partial correctness*: the behavior of fine-grained code is always a *refinement* of the behavior of its type erasure. Either the program produces the same behavior as its type erasure, or it results in a dynamic error. So at this coarse level, most of the program's behavior can be understood in terms of the behavior of dynamically typed code.

As discussed in Chapter 1, we express all reasoning principles as relational principles, which we view as properties of programs "in motion", i.e. properties that state how syntactic changes over the lifespan of the program development process affect the behavior of the program. These two classes of properties are formalized in a similar fashion. We first specify a relation on syntactic types and/or terms, and then we prove that some semantic property is implied by the relation. For equality, if $t \equiv u$ is true then we want the behavior of t and u to be the same. For graduality, if $t \sqsubseteq u$ is true then t errors more, but otherwise behaves the same as u. In both cases, this semantic property is formalized as a *contextual* property: how does swapping out t for u in the context of a larger program change the ultimate behavior?

$$
\begin{array}{rcl}
C & ::= & [\cdot] \mid \mathrm{FromType}^{A} C \mid \mathrm{ToType}^{A} C \mid \lambda x : A.C \mid C\,u \mid t\,C \\
& \mid & (C, t_2) \mid (t_1, C) \mid \pi_i C \\
& \mid & \texttt{if}\ C\ \texttt{then}\ u_t\ \texttt{else}\ u_f \mid \texttt{if}\ t\ \texttt{then}\ C\ \texttt{else}\ u_f \mid \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ C
\end{array}
$$

$$
t \equiv t \qquad \frac{t \equiv u}{u \equiv t} \qquad \frac{t \equiv t' \qquad t' \equiv t''}{t \equiv t''} \qquad \frac{t \equiv u}{C[t] \equiv C[u]}
$$

$$
(\lambda x : A.t)\, v \equiv t[v/x] \qquad (\lambda x.t)\, v \equiv t[v/x] \qquad \frac{\Gamma \vdash v : A \to B}{v \equiv \lambda x : A.v\, x}
$$

$$
\pi_i(v_1, v_2) \equiv v_i \qquad \frac{\Gamma \vdash v : A_1 \times A_2}{v \equiv (\pi_1 v, \pi_2 v)}
$$

$$
\texttt{if true then}\ u_t\ \texttt{else}\ u_f \equiv u_t \qquad \texttt{if false then}\ u_t\ \texttt{else}\ u_f \equiv u_f
$$

$$
\frac{\Gamma \vdash t : A \qquad x : \texttt{Bool} \in \Gamma \qquad \Gamma \vdash v : \texttt{Bool}}{t \equiv \texttt{if}\ v\ \texttt{then}\ t[\texttt{true}/x]\ \texttt{else}\ t[\texttt{false}/x]}
$$

$$
\frac{\Gamma \vdash t \qquad x : \texttt{Bool} \in \Gamma \qquad \Gamma \vdash v : \texttt{Bool}}{t \equiv \texttt{if}\ v\ \texttt{then}\ t[\texttt{true}/x]\ \texttt{else}\ t[\texttt{false}/x]}
$$

Figure 2.7: $\beta\eta$ equality for Multi-Language Terms

2.2.1 *Reasoning about Equality*

First, we discuss reasoning principles concerning when two programs have equivalent behavior, i.e., equational reasoning principles. Specifically,, we discuss *βη equivalence* which tell us in certain cases that programs have *the same* behavior, and so rewriting one as the other results in no observable change.

Figure 2.7 shows call-by-value $\beta\eta$ equivalence for our gradual multilanguage. First, we have generic reasoning principles across different types: reflexivity, symmetry, transitivity, and closure under arbitrary contexts. Of these, context closure is most interesting: it says that if two programs have equivalent behavior, placing them in the same program context produces equivalent behavior. The β laws should be uncontroversial: they are typical operational reductions stated as contextual equivalences, for instance restricting the input of a function to be a value before reducing. In fact, these β laws are contextual equivalences in many *dynamically typed* languages, and so do not say much about type soundness.

The η laws however are not satisfied by dynamic languages, and so can be thought of as *the difference* between reasoning about dynamically

typed programs and statically typed programs. The general idea of an η law is that it says that values of each type are all equivalent to a value produced by an introduction form of that type, and all behavior they exhibit is given by the elimination forms for that type.

in call-by-value languages η applies to values, but pure languages have η for all terms and for call-by-name languages η is for strict *terms*

For example, the η law for functions says that any value of function type is equivalent to one produced by λ by simply applying the original value to the λ-bound variable. If v is a λ function itself, this follows already by β equivalence, but the more useful case is when v is *itself* a variable. This shows that the η principle is really about restricting the power of the context to distinguish between programs: if every variable at function type is equivalent to a λ, that means that the context cannot pass non-function values as inputs.

The η law for strict pairs is of a dual nature: rather than being about terms with an output type of $A_1 \times A_2$, it is concerned with terms that have a *free variable* of type $A_1 \times A_2$. In plain English, it says that any term with a free variable $y : A_1 \times A_2$ is equivalent to one that immediately pattern-matches on y, and then replaces y with the reconstructed pair. This means first and foremost that pattern-matching on y is *safe*: compare to a strongly typed dynamic language where projecting from the pair would result in a runtime type error on non-pair inputs like booleans or functions. Second, it means that there is no more to a value of product type than what its components are, since the value in the continuation of the pattern match is replaced by the reconstructed pair. This rules out things like lazy evaluation, where a pair is a thunk that is forced when a pattern match is performed, and also rules out "junk" in the value like metadata allowing for runtime reflection.

These η principles provide a principled foundation for type-based optimization and refactoring. For instance, suppose you have an input $y : A_1 \times A_2$ and construct a function that pattern matches on y: $\lambda z : B.\texttt{let}\ (x_1, x_2) = y; t$, Upon reviewing the code you might decide that it is clearer to perform the pattern-match before constructing the function, perhaps because the second component of the pair is never used, and you want to reduce the size of the closure at runtime, rewriting it as $\texttt{let}\ (x_1, x_2) = y; (\lambda z : B.t)$. This intuitively obvious optimization is justified by the $\beta\eta$ principles for pairs:

$$\begin{aligned}
\lambda z : B.\texttt{let}\ (x_1, x_2) = y; t &\approx^{\text{ctx}} \texttt{let}\ (x_1', x_2') = y; \lambda z : B.\texttt{let}\ (x_1, x_2) = (x_1', x_2'); t \\
&\qquad\qquad (\times\eta) \\
&\approx^{\text{ctx}} \texttt{let}\ (x_1', x_2') = y; \lambda z : B.t[x_1'/x_2][x_2'/x_1] \\
&\qquad\qquad (\times\beta) \\
&\approx^{\text{ctx}} \texttt{let}\ (x_1, x_2) = y; \lambda z : B.t \qquad (\alpha)
\end{aligned}$$

First, the η principle says you can deconstruct x immediately, and then the β principle simplifies the continuation, and α equivalence fixes the change of variable names. This program equivalence would

not be valid in a dynamic language because it might introduce a runtime error into the program if y is instantiated with a non-pair, you are only justified in making the optimization because the type information $y : A_1 \times A_2$ is reliable.

It might seem difficult for a careful language design to violate these η reasoning principle, but this transformation is invalid in the transient semantics of gradual typing [87]. In transient semantics, only the top-level constructor gives reliable information, with deeper checks performed only as a value is inspected. For instance, in transient semantics, the type $\texttt{Bool} \times \texttt{Int}$ includes values like $(\lambda x.x, \texttt{true})$, which is a pair, but the components of the pair do not match the type. Then η expansion for pairs is not generally valid, because pattern matching on a value like this produces a type error: $\texttt{let}\ (y,z) = (\lambda x.x, \texttt{true}); (y,z)$ runs to a type error reporting that either $\lambda x.x$ is not a boolean or that $\texttt{true}$ is not a number. This limits the flexibility of type-based optimization and refactoring.

To show that $\equiv$ gives a valid reasoning principle, we want to prove that if two closed terms are equivalent $t \equiv u$, then they have the same behavior. More generally, if t and u are open terms, then since $\beta\eta$ equivalence is closed under plugging into the same context, this means that t and u are indistinguishable when put into a larger context. The validity of the $\beta\eta$ equivalence relation is given by the following theorem:

Definition 1. If $\cdot \vdash t : A$ and $\cdot \vdash u : A$ are well-typed cast calculus programs we say they have the same behavior, written $t \simeq u$ if

- $t \mapsto^* v$ and $u \mapsto^* v'$ for some values v, v'
- $t \mapsto^* \mathcal{U}$ and $u \mapsto^* \mathcal{U}$
- $t \Uparrow$ and $v \Uparrow$

Theorem 2. *If $t \equiv u$ is derivable for two closed fine-grained terms t, u, then $t^{fg} \simeq u^{fg}$.*

For multi-language terms,

- *If $t \equiv u$ is derivable for two closed static multilanguage terms t, u, then $t^S \simeq t'^S$.*
- *If $t \equiv u$ is derivable for two closed dynamic multilanguage terms t, u, then $t^D \simeq t'^D$.*

We will prove this in Chapter 3, using some lemmas in §2.2 at the end of this chapter.

2.2.2 *Precision and Graduality*

The next reasoning principle of gradual typing we present is the *graduality principle*, originally called the *gradual guarantee* in [75]. We coined

the name graduality in [56] based on an analogy with the parametricity property of polymorphic calculi. This analogy is formally substantiated in Chapter 10. The graduality principle helps us to reason about the *migration* process from dynamic to static typing. The idea is that migrating to a more statically typed style of programming should have minimal effect on the *behavior* of the program: if the program was working before, then the correctness should not be affected by adding types. To formalize this we need to define what syntactic changes count as migrations. This is a bit simpler to define in the fine-grained syntax, so we define that first. As mentioned above, this also alleviates the programmer of the need to understand all of the details of the runtime semantics of the dynamic type checking, at the coarse-grained level of partial correctness the behavior of the program remains the same when types are added.

Graduality fails when types affect program behavior in ways besides the presence or absence of errors. The original motivation for the property was to formalize why allowing deep runtime type tests in a gradual system could be problematic [11]. A type test is a predicate that asks what type a dynamically typed value has, for instance, integer? and boolean? in Scheme/Racket check for integers and booleans respectively. An example of a deep type test is $(\texttt{Bool} \to \texttt{Bool})?$ which would test if a value is a function taking booleans to booleans. Since such a property is undecidable, the implementation must rely on some syntactic approximation to produce a total predicate, for instance checking if the function was syntactically *written* as a boolean to boolean function. However such a check would violate graduality, as the dynamically typed identity function would fail the test, while a boolean typed variant would:

$$(\texttt{Bool} \to \texttt{Bool})?(\lambda x : ?.x) \mapsto \texttt{false}$$

$$(\texttt{Bool} \to \texttt{Bool})?(\lambda x : \texttt{Bool}.x) \mapsto \texttt{true}$$

These can be avoided by using *shallow* type tests, really *tag* tests. I.e., we can only ask if a value is a function at all, not what its domain and codomain are.

In Figure 2.8 we present two relations, *type precision* and *term precision*. We define $A \sqsubseteq B$, pronounced "A is more precise than B" or "A is less dynamic than B" to hold when A and B are structurally the same except that some subformulae of A correspond to the dynamic type in B. In terms of migration, we think of $A \sqsubseteq B$ as meaning that A is a valid choice of migrating from B to a more precise type.

Next we define *term precision*. Similar to type precision, we say $t \sqsubseteq t'$, pronounced "t is more precise than t'" to mean that t is a valid migration of t' to more precise types. This is related to type precision in that whenever $t \sqsubseteq t'$ holds, the type of t must be more precise than the type of t', and similarly for all of their free variables. The full form of the judgment is then $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$, where Φ is a sequence of

ordering assumptions on free variables $x \sqsubseteq x' : B \sqsubseteq B'$. We maintain the invariant that if $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$, then $\Phi_l \vdash t : A$ and $\Phi_r \vdash t' : A'$ where Φ_l extracts the lower bounds of all the assumptions in Φ and Φ_r the upper bounds. The rules come in two groups: first the cast rules and second the congruence rules. The cast rules say that if $t \sqsubseteq t'$ holds, then adding a cast to either side preserves ordering as long as the side conditions on typing remain valid. In terms of migration, this allows for the addition or removal of type casts, as long as the types are still getting more precise. The remaining rules are the *congruence* rules, one for each type former. For most rules, this simply says that the term constructor is monotone, with the variable and λ rules being the most interesting. First, the variable rule says that if variables are assumed to be in an ordering then they are in fact ordered. Second the λ rule allows for the type annotation on the function to become more precise.

Next in Figures 2.9 and 2.10, we have term precision for our multilanguage syntax. Unlike the fine-grained syntax, our multilanguage syntax does not feature a dynamic type, so there is no non-trivial notion of type precision. Instead, migration goes directly from dynamic typing to a syntactic type. We can write this as a kind of "pseudo-precision" $A \sqsubseteq$ dynamic where dynamic here is not a static type but represents the "pseudo-type" of dynamically typed terms. Term precision can be divided up into 3 different relations. First, we might migrate a dynamically typed term to be statically typed, written $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq$ dynamic, which as a side condition means $\Phi_l \vdash t : A$ and $\Phi_r \vdash t'$. Next, static terms can be a possible migration of a static term, for instance when a subterm migrates from dynamic to static typing. Since type precision is trivial, both terms have the same type, written $\Phi \vdash t \sqsubseteq t' : A$ and having side condition $\Phi_l \vdash t : A$ and $\Phi_r \vdash t' : A$. Finally, a dynamic term can be a possible migration of another dynamic term, written $\Phi \vdash t \sqsubseteq t'$ with side condition $\Phi_l \vdash t$ and $\Phi_r \vdash t'$. The term precision context Φ can have assumptions of these three forms: left typed and right untyped written $x \sqsubseteq x' : A \sqsubseteq$ dyamic, both typed written $x \sqsubseteq x' : A$ and both dynamic written $x \sqsubseteq x'$.

Next, we define the appropriate semantic soundness theorem for migration in each style: the *graduality* principle. The graduality principle says that making a term's types more precise results in only additional error checking, and otherwise does not interfere with behavior.

Definition 3. We define the error ordering $t \sqsubseteq^{\text{err}} u$ for closed cast calculus terms to mean one of the following holds:

- $t \mapsto^* \lightning$.
- $t \mapsto^* v_t$ and $u \mapsto^* v_u$
- $t \Uparrow$ and $u \Uparrow$

$$A \sqsubseteq ? \qquad \texttt{Bool} \sqsubseteq \texttt{Bool} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \times B \sqsubseteq A' \times B'}$$

$$\Phi ::= \cdot \mid \Phi, x \sqsubseteq x' : A \sqsubseteq A'$$

$$\begin{aligned} \cdot_l &= \cdot \\ (\Phi, x \sqsubseteq x' : A \sqsubseteq A')_l &= \Phi_l, x : A \end{aligned}$$

$$\begin{aligned} \cdot_r &= \cdot \\ (\Phi, x \sqsubseteq x' : A \sqsubseteq A')_l &= \Phi_r, x' : A' \end{aligned}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad A \sqsubseteq B'}{\Phi \vdash t \sqsubseteq t' :: B' : A \sqsubseteq B'} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad B \sqsubseteq A'}{\Phi \vdash t :: B \sqsubseteq t' : B \sqsubseteq A'}$$

$$\frac{x \sqsubseteq A \sqsubseteq A' \in \Phi}{\Phi \vdash x \sqsubseteq x' : A \sqsubseteq A'}$$

$$\frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B'}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x' : A'.t' : A \to B \sqsubseteq A' \to B'}$$

$$\frac{\begin{array}{cc} \Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' & \Phi \vdash u \sqsubseteq u' : B \sqsubseteq B' \\ \mathrm{dom}(A) \sim B & \mathrm{dom}(A') \sim B' \end{array}}{\Phi \vdash t\, u \sqsubseteq t'\, u' : \mathrm{cod}(A) \sqsubseteq \mathrm{cod}(A')}$$

$$\frac{\forall i \in \{1,2\}.\ \Phi \vdash t_i \sqsubseteq t_i' : A_i \sqsubseteq A_i'}{\Phi \vdash (t_1, t_2) \sqsubseteq (t_1', t_2') : A_1 \times A_2 \sqsubseteq A_1' \times A_2'}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'}{\Phi \vdash \pi_i t \sqsubseteq \pi_i t' : \pi_i(A) \sqsubseteq \pi_i(A')} \qquad \Phi \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool} \sqsubseteq \texttt{Bool}$$

$$\Phi \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool} \sqsubseteq \texttt{Bool}$$

$$\frac{\begin{array}{ccc} \Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' & A \sim \texttt{Bool} & A' \sim \texttt{Bool} \\ \Phi \vdash u_t \sqsubseteq u_t' : B_t \sqsubseteq B_t' & \Phi \vdash u_f \sqsubseteq u_f' : B_f \sqsubseteq B_f' & \end{array}}{\Phi \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f \sqsubseteq \texttt{if}\ t'\ \texttt{then}\ u_t'\ \texttt{else}\ u_f' : B_t \sqcap B_f \sqsubseteq B_t' \sqcap B_f'}$$

Figure 2.8: Type and Term Precision for Fine-Grained Terms

$$\frac{\Phi \vdash t \sqsubseteq t'}{\Phi \vdash \text{FromType}^{A} t \sqsubseteq t' : A \sqsubseteq \text{dynamic}} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq \text{dynamic}}{\Phi \vdash t \sqsubseteq \text{FromType}^{A} t' : A}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A}{\Phi \vdash t \sqsubseteq \text{ToType}^{A'} t' : A \sqsubseteq \text{dynamic}} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq \text{dynamic}}{\Phi \vdash \text{ToType}^{A} t \sqsubseteq t'}$$

$$\frac{x \sqsubseteq x' : A \in \Phi}{\Phi \vdash x \sqsubseteq x' : A} \qquad \frac{x \sqsubseteq x' : A \sqsubseteq \text{dynamic} \in \Phi}{\Phi \vdash x \sqsubseteq x' : A \sqsubseteq \text{dynamic}} \qquad \frac{x \sqsubseteq x' \in \Phi}{\Phi \vdash x \sqsubseteq x'}$$

$$\frac{\Phi, x \sqsubseteq x' : A' \vdash t \sqsubseteq t' : B}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x' : A.t' : A \to B}$$

$$\frac{\Phi, x : A \sqsubseteq x' \,\text{dynamic} \vdash t \sqsubseteq t' : B \sqsubseteq \text{dynamic}}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x'.t' : A \to B \sqsubseteq \text{dynamic}}$$

$$\frac{\Phi, x \,\text{dynamic} \sqsubseteq x' \,\text{dynamic} \vdash t \sqsubseteq t' : B \sqsubseteq \text{dynamic}}{\Phi \vdash \lambda x.t \sqsubseteq \lambda x'.t'}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \to B \qquad \Phi \vdash u \sqsubseteq u' : A}{\Phi \vdash t\, u \sqsubseteq t'\, u' : B}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \to B \sqsubseteq \text{ dynamic} \qquad \Phi \vdash u \sqsubseteq u' : A \sqsubseteq \text{ dynamic}}{\Phi \vdash t\, u \sqsubseteq t'\, u' : B \sqsubseteq \text{ dynamic}}$$

$$\frac{\Phi \vdash t \sqsubseteq t' \qquad \Phi \vdash u \sqsubseteq u'}{\Phi \vdash t\, u \sqsubseteq t'\, u'}$$

Figure 2.9: Boundary, Variable and Function Term Precision for Multi-language Terms

$$\frac{\forall i \in \{1,2\}.\ \Phi \vdash t_i \sqsubseteq t_i' : A_i}{\Phi \vdash (t_1, t_2) \sqsubseteq (t_1', t_2') : A_1 \times A_2}$$

$$\frac{\forall i \in \{1,2\}.\ \Phi \vdash t_i \sqsubseteq t_i' : A_i \sqsubseteq \text{dynamic}}{\Phi \vdash (t_1, t_2) \sqsubseteq (t_1', t_2') : A_1 \times A_2 \sqsubseteq \text{dynamic}}$$

$$\frac{\forall i \in \{1,2\}.\ \Phi \vdash t_i \sqsubseteq t_i'}{\Phi \vdash (t_1, t_2) \sqsubseteq (t_1', t_2')} \qquad \frac{\Phi \vdash t \sqsubseteq t : A_1 \times A_2}{\Phi \vdash \pi_i t \sqsubseteq \pi_i t' : A_i}$$

$$\frac{\Phi \vdash t \sqsubseteq t : A_1 \times A_2 \sqsubseteq \text{dynamic}}{\Phi \vdash \pi_i t \sqsubseteq \pi_i t' : A_i \sqsubseteq \text{dynamic}} \qquad \frac{\Phi \vdash t \sqsubseteq t}{\Phi \vdash \pi_i t \sqsubseteq \pi_i t'}$$

$$\Phi \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool} \qquad \Phi \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool} \sqsubseteq \text{dynamic}$$

$$\Phi \vdash \texttt{true} \sqsubseteq \texttt{true} \qquad \Phi \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool}$$

$$\Phi \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool} \sqsubseteq \text{dynamic} \qquad \Phi \vdash \texttt{false} \sqsubseteq \texttt{false}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : \texttt{Bool} \qquad \Phi \vdash u_t \sqsubseteq u_t' : B \qquad \Phi \vdash u_f \sqsubseteq u_f' : B}{\Phi \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f \sqsubseteq \texttt{if}\ t'\ \texttt{then}\ u_t'\ \texttt{else}\ u_f' : B}$$

$$\frac{\begin{array}{c}\Phi \vdash t \sqsubseteq t' : \texttt{Bool} \sqsubseteq \text{dynamic} \\ \Phi \vdash u_t \sqsubseteq u_t' : B \sqsubseteq \text{dynamic} \qquad \Phi \vdash u_f \sqsubseteq u_f' : B \sqsubseteq \text{dynamic}\end{array}}{\Phi \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f \sqsubseteq \texttt{if}\ t'\ \texttt{then}\ u_t'\ \texttt{else}\ u_f' : B \sqsubseteq \text{dynamic}}$$

$$\frac{\Phi \vdash t \sqsubseteq t' \qquad \Phi \vdash u_t \sqsubseteq u_t' \qquad \Phi \vdash u_f \sqsubseteq u_f'}{\Phi \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f \sqsubseteq \texttt{if}\ t'\ \texttt{then}\ u_t'\ \texttt{else}\ u_f'}$$

Figure 2.10: Product, Boolean Term Precision for Multi-language Terms

Theorem 4 (Graduality). *If t, u are fine-grained terms satisfying $\cdot \vdash t \sqsubseteq u : A \sqsubseteq B$, then $t^{fg} \sqsubseteq^{err} u^{fg}$.*

For multilanguage terms,

- *If $\cdot \vdash t \sqsubseteq u : A$ then $t^S \sqsubseteq^{err} u^S$.*
- *If $\cdot \vdash t \sqsubseteq u : A \sqsubseteq$ dynamic then $t^S \sqsubseteq^{err} u^D$.*
- *If $\cdot \vdash t \sqsubseteq u$ then $t^D \sqsubseteq^{err} u^D$.*

2.2.3 *Reducing Surface Reasoning to Cast Calculus Reasoning*

To give a modular development and abstract away from specific idiosyncrasies of surface syntax, in Chapter 3 we will focus our attention on semantics of the *cast calculus*. To facilitate this, we develop *sufficient* conditions on the semantics of the cast calculus in order to prove soundness of $\beta\eta$ equivalence and graduality for the surface languages. To do this we define the analogues of the syntactic $\beta\eta$ equivalence and term precision, and prove (1) that elaboration preserves the syntactic relation and (2) the relation for cast calculus terms implies the desired semantic property. Since the soundness theorems are stated in terms of the cast calculus operational semantics, we don't need any kind of adequacy or simulation theorem.

We define $\beta\eta$ the same way as for the surface language, adding casts to our notion of context.

Lemma 5. *v^{fg}, v^S and v^D are all values.*

Proof. Clear by induction. For fine-grained terms, note that it follows because only elimination forms introduce casts. □

Lemma 6.

- $t[v/x]^{fg} = t^{fg}[v^{fg}/x]$
- $t[v/x]^S = t^S[v^S/x]$ *if* $x : A$
- $t[v/x]^S = t^S[v^D/x]$ *if* x *dynamic*
- $t[v/x]^D = t^S[v^S/x]$ *if* $x : A$
- $t[v/x]^D = t^S[v^D/x]$ *if* x *dynamic*

Proof. Clear by induction on t. □

Lemma 7. *For fine-grained terms, if $t \equiv u$ then $t^{fg} \equiv u^{fg}$.*

For multilanguage terms

- *For typed terms, if $t \equiv u$ then $t^S \equiv u^S$.*
- *For dynamic terms, if $t \equiv u$ then $t^D \equiv u^D$.*

Proof. Most cases are direct by induction, including all multilanguage cases. We show the two most interesting fine-grained cases.

$$
\begin{aligned}
C \quad ::= {} & [\cdot] \mid \langle A \Leftarrow B \rangle C \mid \lambda x : A.C \mid C\,u \mid t\,C \\
& \mid (C, t_2) \mid (t_1, C) \mid \pi_i C \\
& \mid \texttt{if}\ C\ \texttt{then}\ u_t\ \texttt{else}\ u_f \mid \texttt{if}\ t\ \texttt{then}\ C\ \texttt{else}\ u_f \mid \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ C
\end{aligned}
$$

$$
t \equiv t \qquad \frac{t \equiv u}{u \equiv t} \qquad \frac{t \equiv t' \qquad t' \equiv t''}{t \equiv t''} \qquad \frac{t \equiv u}{C[t] \equiv C[u]}
$$

$$
(\lambda x : A.t)\, v \equiv t[v/x] \qquad (\lambda x.t)\, v \equiv t[v/x] \qquad \frac{\Gamma \vdash v : A \to B}{v \equiv \lambda x : A.v\, x}
$$

$$
\pi_i(v_1, v_2) \equiv v_i \qquad \frac{\Gamma \vdash v : A_1 \times A_2}{v \equiv (\pi_1 v, \pi_2 v)}
$$

$$
\texttt{if true then}\ u_t\ \texttt{else}\ u_f \equiv u_t \qquad \texttt{if false then}\ u_t\ \texttt{else}\ u_f \equiv u_f
$$

$$
\frac{\Gamma \vdash t : A \qquad x : \texttt{Bool} \in \Gamma \qquad \Gamma \vdash v : \texttt{Bool}}{t \equiv \texttt{if}\ v\ \texttt{then}\ t[\texttt{true}/x]\ \texttt{else}\ t[\texttt{false}/x]}
$$

$$
\frac{\Gamma \vdash t \qquad x : \texttt{Bool} \in \Gamma \qquad \Gamma \vdash v : \texttt{Bool}}{t \equiv \texttt{if}\ v\ \texttt{then}\ t[\texttt{true}/x]\ \texttt{else}\ t[\texttt{false}/x]}
$$

$$
t \equiv \langle A \Leftarrow A \rangle t
$$

Figure 2.11: $\beta\eta$ Equality for Cast Calculus Terms

- Case $\Gamma \vdash v : A \to B$, then $v \equiv \lambda x : A.v\, x$. We need to show that

  $$v^{fg} \equiv (\lambda x : A.v\, x)^{fg}$$

  which after expanding the definition of the translation means proving

  $$v^{fg} \equiv \lambda x : A.v^{fg}\, (\langle A \Leftarrow A \rangle x)$$

  Which because v^{fg} is a value (Lemma 5), η in the cast calculus and the identity cast rule.

- Case $\Gamma \vdash t : A$ and $x : \texttt{Bool} \in \Gamma$, then $t^{fg} \equiv (\texttt{if}\ x\ \texttt{then}\ t[\texttt{true}/x]\ \texttt{else}\ t[\texttt{false}/x])^{fg}$. Simplifying the right we need to prove

$$t^{fg} \equiv \texttt{if}\ x\ \texttt{then}\ \langle A \Leftarrow A \rangle (t[\texttt{true}/x])^{fg}\ \texttt{else}\ \langle A \Leftarrow A \rangle (t[\texttt{false}/x])^{fg}$$

  By η for booleans in the cast calculus, we know

  $$t^{fg} \equiv \texttt{if}\ x\ \texttt{then}\ t^{fg}[\texttt{true}/x]\ \texttt{else}\ t^{fg}[\texttt{false}/x]$$

  So we need to show each branch of the if is equivalent. We cover the true case, the other is analogous. We need to show

  $$t^{fg}[\texttt{true}/x] \equiv \langle A \Leftarrow A \rangle (t[\texttt{true}/x])^{fg}$$

  First, $t^{fg}[\texttt{true}/x] \equiv t[\texttt{true}/x]^{fg}$ by Lemma 6. Then the rest follows by the identity cast rule.

□

We need a similar theorem for term precision.

Lemma 8. *p If $\Phi \vdash t \sqsubseteq u : A \sqsubseteq A'$ then $\Phi \vdash t^{fg} \sqsubseteq u^{fg} : A \sqsubseteq A'$*
For multi-language terms,

- *If $\Phi \vdash t \sqsubseteq u : A$, then $\Phi^{ml} \vdash t^S \sqsubseteq u^S : A \sqsubseteq A$*
- *If $\Phi \vdash t \sqsubseteq u : A \sqsubseteq$ dynamic, then $\Phi^{ml} \vdash t^S \sqsubseteq u^D : A \sqsubseteq ?$*
- *If $\Phi \vdash t \sqsubseteq u$, then $\Phi^{ml} \vdash t^D \sqsubseteq u^D : ? \sqsubseteq ?$*

Proof. Most cases are straightforward by induction, we show the cases that involve casts. Interestingly, the fine-grained cases for elimination forms naturally break into 3 cases: both static, left static and right dynamic and finally both dynamic, which mirrors the multilanguage relation.

First, for fine-grained terms

- $$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad A \sqsubseteq B'}{\Phi \vdash t \sqsubseteq t' :: B' : A \sqsubseteq B'}$$

  We need to show

  $$\Phi \vdash t^{fg} \sqsubseteq \langle B' \Leftarrow B \rangle t'^{fg}$$

  which follows by inductive hypothesis and the cast rule.

- $\dfrac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad \Phi \vdash t \sqsubseteq u' : B \sqsubseteq B' \quad \mathrm{dom}(A) \sim B \qquad \mathrm{dom}(A') \sim B'}{\Phi \vdash t\, u \sqsubseteq t'\, u' : \mathrm{cod}(A) \sqsubseteq \mathrm{cod}(A')}$

  We need to show

$$(? \to ? \between t^{fg})\,(\langle \mathrm{dom}(A) \Leftarrow B \rangle u^{fg}) \sqsubseteq (? \to ? \between t'^{fg})\,(\langle \mathrm{dom}(A') \Leftarrow B' \rangle u'^{fg})$$

  There are three cases to consider: either both A and A' are function types, A is a function type and $A' = ?$, or both are $?$.

  - If $A = A_i \to A_o$ and $A' = A'_i \to A'_o$, then we need to show

$$\Phi \vdash t^{fg}\,(\langle A_i \Leftarrow B \rangle u^{fg}) \sqsubseteq t'^{fg}\,(\langle A'_i \Leftarrow B' \rangle u'^{fg}) : A_o \sqsubseteq A'_o$$

    First, by the application rule it is sufficient to show the functions and their arguments are related.

    * $t^{fg} \sqsubseteq t'^{fg}$ holds by inductive hypothesis.
    * $\langle A_i \Leftarrow B \rangle u^{fg} \sqsubseteq \langle A'_i \Leftarrow B' \rangle u'^{fg}$ follows which follows by cast monotonicity and inductive hypothesis.

  - If $A = A_i \to A_o$ and $A' = ?$, then we need to show

$$\Phi \vdash t^{fg}\,(\langle A_i \Leftarrow B \rangle u^{fg}) \sqsubseteq (\langle ? \to ? \Leftarrow ? \rangle t'^{fg})\,(\langle ? \Leftarrow B' \rangle u'^{fg}) : A_o \sqsubseteq ?$$

    Using application monotonicity, it is sufficient to show the functions and their arguments are related

    * $t^{fg} \sqsubseteq \langle ? \to ? \Leftarrow ? \rangle t'^{fg}$ follows by the cast right rule and inductive hypothesis.
    * $\langle A_i \Leftarrow B \rangle u^{fg} \sqsubseteq \langle ? \Leftarrow B' \rangle u'^{fg}$ follows by cast monotonicity and the inductive hypothesis.

  - If $A = A' = ?$, then we need to show

$$\Phi \vdash (\langle ? \to ? \Leftarrow ? \rangle t^{fg})\,(\langle ? \Leftarrow B \rangle u^{fg}) \sqsubseteq (\langle ? \to ? \Leftarrow ? \rangle t'^{fg})\,(\langle ? \Leftarrow B' \rangle u'^{fg}) : ? \sqsubseteq ?$$

    Using function application monotonicity it is sufficient to show functions and arguments are related.

    * $\langle ? \to ? \Leftarrow ? \rangle t^{fg} \sqsubseteq \langle ? \to ? \Leftarrow ? \rangle t'^{fg}$ follows by cast monotonicity and inductive hypothesis.
    * $\langle ? \Leftarrow B \rangle u^{fg} \sqsubseteq \langle ? \Leftarrow B \rangle u'^{fg}$ follows by cast monotonicity and inductive hypothesis as well.

Next, for multi-language terms the most interesting cases are the import rules and the rules where one side is static and the other isn't.

- 4 import rules $\dfrac{\Phi \vdash t \sqsubseteq t'}{\Phi \vdash \mathrm{FromType}^{A} t \sqsubseteq t' : A \sqsubseteq \mathrm{dynamic}}$

  In translation, we have by inductive hypothesis that $\Phi^{ml} \vdash t^{D} \sqsubseteq t'^{D} : ? \sqsubseteq ?$ and need to show $\Phi^{ml} \vdash \langle A \Leftarrow ? \rangle t \sqsubseteq t'^{D} : A \sqsubseteq ?$ which follows directly from a cast rule since $A \sqsubseteq ?$. The other cast rules are similar.

- $$\frac{\Phi, x : A \sqsubseteq x' \text{ dynamic} \vdash t \sqsubseteq t' : B \sqsubseteq \text{dynamic}}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x'.t' : A \to B \sqsubseteq \text{dynamic}}$$

  By inductive hypothesis we know $\Phi^{ml}, x \sqsubseteq x' : A \sqsubseteq ? \vdash t^S \sqsubseteq t'^D : B \sqsubseteq ?$ and we need to show $\Phi^{ml} \vdash \lambda x : A.t^S \sqsubseteq \langle ? \Leftarrow ? \to ? \rangle (\lambda x : ?.t'^D) : A \to B \sqsubseteq ?$.

  This follows from the cast rule, and inductive hypothesis.

- $$\frac{\Phi \vdash t \sqsubseteq t' : A \to B \sqsubseteq \text{ dynamic} \qquad \Phi \vdash u \sqsubseteq u' : A \sqsubseteq \text{ dynamic}}{\Phi \vdash t\, u \sqsubseteq t'\, u' : B \sqsubseteq \text{ dynamic}}$$

  We need to show

  $$\Phi^{ml} \vdash t^S\, u^S \sqsubseteq (\langle ? \to ? \Leftarrow ? \rangle t^D)\, u^D : B \sqsubseteq ?$$

  Which follows from inductive hypothesis and one cast right rule.n

□

In the next chapter, we'll establish that syntactic $\equiv$ and $\sqsubseteq$ imply our desired semantic properties $\simeq$ and $\sqsubseteq^{\text{err}}$.

$$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad B \sqsubseteq B'}{\Phi \vdash \langle B \Leftarrow A \rangle t \sqsubseteq \langle B \Leftarrow A' \rangle t' : B \sqsubseteq B'}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad B \sqsubseteq A'}{\Phi \vdash \langle B \Leftarrow A \rangle t \sqsubseteq t' : B \sqsubseteq A'} \qquad \frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad A \sqsubseteq B'}{\Phi \vdash t \sqsubseteq \langle B' \Leftarrow A' \rangle t' : A \sqsubseteq B'}$$

$$\frac{x \sqsubseteq x' : A \sqsubseteq A' \in \Phi}{\Phi \vdash x \sqsubseteq x' : A \sqsubseteq A'}$$

$$\frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B'}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x' : A'.t' : A \to B \sqsubseteq A' \to B'}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \to B \sqsubseteq A' \to B' \qquad \Phi \vdash u \sqsubseteq u' : A \sqsubseteq A'}{\Phi \vdash t\, u \sqsubseteq t'\, u' : B \sqsubseteq B'}$$

$$\frac{\forall i \in \{1,2\}.\ \Phi \vdash t_i \sqsubseteq t_i' : A_i \sqsubseteq A_i'}{\Phi \vdash (t_1, t_2) \sqsubseteq (t_1', t_2') : A_1 \times A_2 \sqsubseteq A_1' \times A_2'}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A_1 \times A_2 \sqsubseteq A_1' \times A_2'}{\Phi \vdash \pi_i t \sqsubseteq \pi_i t' : A_i \sqsubseteq A_i'} \qquad \Phi \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool} \sqsubseteq \texttt{Bool}$$

$$\Phi \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool} \sqsubseteq \texttt{Bool}$$

$$\frac{\begin{array}{c}\Phi \vdash t \sqsubseteq t' : \texttt{Bool} \sqsubseteq \texttt{Bool} \\ \Phi \vdash u_t \sqsubseteq u_t' : B \sqsubseteq B' \qquad \Phi \vdash u_f \sqsubseteq u_f' : B \sqsubseteq B'\end{array}}{\Phi \vdash \texttt{if}\ t\ \texttt{then}\ u_t\ \texttt{else}\ u_f \sqsubseteq \texttt{if}\ t'\ \texttt{then}\ u_t'\ \texttt{else}\ u_f' : B \sqsubseteq B'}$$

Figure 2.12: Term Precision for Cast Calculus

3

GRADUALITY FROM EMBEDDING PROJECTION PAIRS

In the previous section we showed some common approaches to syntax and operational semantics of a gradually typed language, and some desirable relational reasoning principles that codify type-based reasoning and migrational reasoning. The remainder of this dissertation will be concerned with the two related questions of (1) how to prove these reasoning principles are valid and (2) how to design a gradual language to ensure these principles hold. This chapter presents the fundamental semantic technique of this work: the study of gradual typing casts using the mathematical concept of *embedding-projection pairs* and how this provides compositional semantic principles for establishing graduality. The main results we will derive using this technique in this chapter will be to prove the validity of $\beta\eta$ equivalence and of graduality for the cast calculus in the previous section, which as shown in §2.2.3 implies analogous theorems about surface calculi.

Our approach is to introduce yet another calculus as a target of elaboration of the cast calculus, but this time the calculus will be a standard statically-typed λ calculus featuring recursive types serving as a semantic metalanguage. Then we will define a simple, modular interpretation of the types, terms and type and term precision as *semantic* properties of types and terms in our metalanguage. The central component of this interpretation is the semantic interpretation of *type precision*, which is interpreted as giving a formal syntax for certain fundamental casts. In this calculus we will interpret the built-in dynamic type as a recursive type, and the built-in type casts as ordinary terms. We then establish a simple *adequacy* theorem that allows us to verify operational properties of our cast calculus by reasoning about the translation to our semantic metalanguage. The $\beta\eta$ equational principles then hold in the source language because they hold in our target calculus, which features the same type structure. To reason about the casts, we show that the somewhat cumbersome operational semantics of the cast calculus is simply explained by decomposition into certain casts that satisfy the property of being *embedding-projection pairs*. Furthermore, these embedding-projection pairs can be defined by induction over *type precision* derivations, providing a *semantic* meaning to our notion of type migration.

This role could also be filled by a denotational semantics.

We summarize the construction now. First, we define an interpretation of values of the dynamic type as a recursive sum of all the "tag types":

$$? = \mu X.\texttt{Bool} + (? \times ?) + (? \to ?)$$

This chapter is based on the paper "Graduality from Embedding-projection Pairs" published at ICFP 2018.

Syntax	Semantics
gradual type A	logical type $\lvert A \rvert$ with ep pair $E_{e,A}, E_{p,A} : \lvert A \rvert \lhd \lvert ? \rvert$
cast $A \Rightarrow B$	$E_{p,B}[E_{e,A}]$
$A \sqsubseteq B$	ep pair $E_{e,A,B}, E_{p,A,B} : \lvert A \rvert \lhd \lvert B \rvert$ satisfying $E_{e,B}[E_{e,A,B}] \approx^{\log} E_{e,A}$ and $E_{p,A,B}[E_{p,B}] \approx^{\log} E_{p,A}$
term $\Gamma \vdash t : A$	$\lvert \Gamma \rvert \vdash [\![t]\!] : \lvert A \rvert$
term precision $\Phi \vdash t \sqsubseteq t' : A$	$\texttt{let}\ E_{e,\Gamma,\Gamma'}\Gamma = \Gamma'; E_{p,A,A'}[t] \sqsubseteq^{\log} t' : A'$

Figure 3.1: Overview of Embedding-Projection Pair Semantics

Next, each cast calculus type A defines a triple $(\lvert A \rvert, E_{e,A}, E_{p,A})$ of

1. A "logical type" $\lvert A \rvert$ in the metalanguage that classifies the types values.

2. An embedding-projection pair $(E_{e,A}, E_{p,A}) : A \lhd ?$, where $E_{e,A}$ says how to embed a value of $\lvert A \rvert$ into ? and vice-versa, $E_{p,A}$ says how to *force* a value of type ? to behave as an $\lvert A \rvert$ value, "projecting out" the A behavior from the ? value.

The embedding and projection correspond precisely to the two boundary forms of the multilanguage syntax of gradual typing: the embedding is the semantics of the boundary importing statically typed values into dynamically typed code, and the projection the opposite boundary importing dynamically typed values into statically typed code. In fact, they also give us enough information to define a semantics of all casts in the cast calculus, by defining the translation of an arbitrary cast as an embedding followed by a projection:

$$[\![\langle B \Leftarrow A \rangle M]\!] = E_{p,B}[E_{e,A}[[\![M]\!]]]$$

That is, to cast from a type A to a type B, first interpret the $\lvert A \rvert$ value as a dynamic value, and then *enforce* the type $\lvert B \rvert$ on the result. Next, we interpret $A \sqsubseteq B$ as saying that there exists an embedding-projection pair $E_{e,A,B}, E_{p,A,B} : A \lhd B$, that exhibits A's embedding (respectively projection) as factorizing through Bs embedding (projection). That is we show that embedding $\lvert A \rvert$ into ? is equivalent to first embedding $\lvert A \rvert$ into $\lvert B \rvert$ and then embedding $\lvert B \rvert$ into ?, and a dual situation for the projection. This captures an informal intuition that when A is more precise than B, enforcing A requires enforcement of B. Furthermore, we give a refined analysis of the structure of $A \sqsubseteq B$ *proofs* and show that the proof rules can be interpreted as compositional rules for constructing embedding-projection pairs.

Finally, we have term precision $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$. Our intuitive idea of precision is that t is the result of migrating t' to have more precise type information, and so we want to know what the effect on program behavior is if we swap t in for t' in the context of a larger

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & \mu\alpha.\, A \mid \alpha \mid 1 \mid A \times B \mid A + B \mid A \to B \\
\text{Terms} & t, s & ::= & \mathcal{U} \mid x \mid \text{let}\, x = t\, \text{in}\, s \mid \text{roll}_A\, t \mid \text{unroll}\, t \mid \langle\rangle \mid \langle t, s\rangle \\
& & \mid & \text{let}\, \langle x, y\rangle = t\, \text{in}\, s \mid \text{inj}_1\, t \mid \text{inj}_2\, t \\
& & \mid & \text{case}\, t\, \text{of}\ \text{inj}_1\, x_1.\, t_1 \mid \text{inj}_2\, x_2.\, t_2 \mid \lambda(x : A).\, t \mid t\ s \\
\text{Values} & v & ::= & x \mid \text{roll}_A\, v \mid \langle\rangle \mid \langle v, v\rangle \mid \text{inj}_1\, v \mid \text{inj}_2\, v \mid \lambda(x : A).\, t \\
\text{Eval. Contexts} & E & ::= & [\cdot] \mid \text{let}\, x = E\, \text{in}\, s \mid \text{roll}_A\, E \mid \text{unroll}\, E \\
& & \mid & \langle E, t\rangle \mid \langle v, E\rangle \mid \text{let}\, \langle x, y\rangle = E\, \text{in}\, s \\
& & \mid & \text{inj}_1\, E \mid \text{inj}_2\, E \mid \text{case}\, E\, \text{of}\ \text{inj}_1\, x_1.\, t_1 \mid \text{inj}_2\, x_2.\, t_2 \\
& & \mid & E\ s \mid v\ E \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Substitutions} & \gamma & ::= & \cdot \mid \gamma, v/x
\end{array}
$$

Figure 3.2: $\lambda_{T,\mathcal{U}}$ Syntax

program. Semantically we want the error approximation ordering that is needed for the graduality theorem to hold. However, these are closed programs, so we need to account for the fact that the inputs should be assumed to be in an ordering relationship. To formalize this, we can add casts to the output of t and its inputs (i.e., free variables) so that it has the same type as t' and then say that they are in an open relationship:

$$\Phi_r \vdash \texttt{let}\ E_{e,\Phi_l,\Phi_r}\Phi_l = \Phi_r; E_{p,A,A'}[t] \sqsubseteq^{\log} t' : A'$$

Where $\texttt{let}\ E_{e,\Phi_l,\Phi_r}\Phi_l = \Phi_r;$ is shorthand for a sequence of $\texttt{let}\ E_{e,A,A'}x = x;$ for each $x : A \sqsubseteq A'$ in Φ. We then define this error approximation relation on open terms as a logical relation that implies that for closed terms implies our semantic notion of graduality: if $t \sqsubseteq^{\log} t'$ then either t errors, in which case t' may have arbitrary behavior, or both reduce to values or both diverge.

3.1 A TYPED METALANGUAGE

The typed language we will translate into is $\lambda_{T,\mathcal{U}}$, a call-by-value typed lambda calculus with iso-recursive types and an uncatchable error. Figure 3.2 shows the syntax of the language. Figure 3.3 shows some of the typing rules; the rest are completely standard.

The types of the language are similar to the cast calculus: they include the standard type formers of products, sums, and functions. Rather than the specific dynamic type, we include the more general, but standard, iso-recursive type $\mu\alpha.\, A$, which is isomorphic to the unfolding $A[\mu\alpha.\, A/\alpha]$ by the terms $\text{roll}_{\mu\alpha.A}\, \cdot$ and $\text{unroll}\, \cdot$. As in the source language we have an uncatchable error $\mathcal{U}$.

$$\boxed{\Gamma \vdash t : A}$$

$$\frac{}{\Gamma \vdash \mho : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash s : B}{\Gamma \vdash \mathsf{let}\, x = t \,\mathsf{in}\, s : B}$$

$$\frac{\Gamma \vdash t : A[\mu\alpha.\, A/\alpha]}{\Gamma \vdash \mathsf{roll}_{\mu\alpha.A}\, t : \mu\alpha.\, A} \qquad \frac{\Gamma \vdash t : \mu\alpha.\, A}{\Gamma \vdash \mathsf{unroll}\, t : A[\mu\alpha.\, A/\alpha]} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle t, s\rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash s : B}{\Gamma \vdash \mathsf{let}\, \langle x, y\rangle = t \,\mathsf{in}\, s : B}$$

$$\frac{\Gamma \vdash t : A'}{\Gamma \vdash \mathsf{inj}_2\, t : A + A'} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x\ :\ A).\, t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash t\, s : B}$$

Figure 3.3: $\lambda_{T,\mho}$ Typing Rules

$$E[\mho] \mapsto^0 \mho \qquad\qquad E[\mathsf{let}\, x = v \,\mathsf{in}\, s] \mapsto^0 E[s[v/x]]$$

$$E[\mathsf{unroll}\, (\mathsf{roll}_A\, v)] \mapsto^1 E[v]$$

$$E[\mathsf{let}\, \langle x_1, x_2\rangle = \langle v_1, v_2\rangle \,\mathsf{in}\, t] \mapsto^0 E[t[v_1/x_1, v_2/x_2]]$$

$$E[(\lambda(x\ :\ A).\, t)\, v] \mapsto^0 E[t[v/x]]$$

$$E[\mathsf{case}\, (\mathsf{inj}_1\, v)\ \mathsf{of}\ \ \mathsf{inj}_1\, x.\, t \mid \mathsf{inj}_2\, x'.\, t'] \mapsto^0 E[t[v/x]]$$

$$E[\mathsf{case}\, (\mathsf{inj}_2\, v)\ \mathsf{of}\ \ \mathsf{inj}_1\, x.\, t \mid \mathsf{inj}_2\, x'.\, t'] \mapsto^0 E[t'[v/x']]$$

$$\frac{}{t \mapsto^0 t} \qquad\qquad \frac{t \mapsto^i t' \qquad t' \mapsto^j t''}{t \mapsto^{i+j} t''}$$

Figure 3.4: $\lambda_{T,\mho}$ Operational Semantics

Figure 3.4 presents the operational semantics of the language. For the purposes of later defining a step-indexed logical relation, we assign a weight to each small step of the operational semantics that is 1 for unrolling a value of recursive type and 0 for other reductions. We then define a "quantitative" reflexive, transitive closure of the small-step relation $t \mapsto^i t'$ that adds the weights of its constituent small steps. When the number of steps doesn't matter, we just use $\mapsto$ and $\mapsto^*$. We can then establish some simple facts about this operational semantics.

Lemma 9 (Subject Reduction). *If $\cdot \vdash t : A$ and $t \mapsto^* t'$ then $\cdot \vdash t' : A$.*

Lemma 10 (Progress). *If $\cdot \vdash t : A$ and t is not a value or $\mathcal{O}$, then there exists t' with $t \mapsto t'$.*

Proof. By induction on the typing derivation for t. □

Lemma 11 (Determinism). *If $t \mapsto s$ and $t \mapsto s'$, then $s = s'$.*

3.2 TRANSLATING GRADUAL TYPING

Next we give our compositional semantics of the cast calculus via translation to the metalanguage. To ensure that we can use the translated terms to reason about the cast calculus terms, we need to prove an *adequacy* theorem that says that if the translation of a term terminates (or errors or diverges), then so does the original cast calculus term. To make this easier, we will provide two, observationally equivalent, translations that differ only in their interpretation of casts. The first translation uses the embedding-projection pairs and is easily amenable to proving graduality. The second translation more directly implements the behavior of the cast calculus and so is easier to establish an adequacy theorem. We will then prove that the two are equivalent.

We start by defining the semantics of gradual types as described in the previous section: a logical type with an ep pair to the dynamic type. In fact, in defining this ep pair, it is simpler to define the semantics of type precision as well, since the ep pair for each type is simply the one corresponding to the proof $A \sqsubseteq ?$ that holds for each type. First, we define the "logical type" $|A|$ for each type in Figure 3.5. Booleans are encoded as $1 + 1$, functions and products are translated to themselves and the dynamic type is a recursive sum of the interpretations of the three tag types.

3.2.1 *Constructive Type Precision*

Next we define a *constructive* interpretation of type precision $A \sqsubseteq B$ as defining an embedding-projection pair (ep pair).

Formally, we define an ep pair as follows:

$$
\begin{array}{rcl}
|?| & \stackrel{\text{def}}{=} & \mu\alpha.\,(1+1)+(\alpha\times\alpha)+(\alpha\to\alpha) \\
|\texttt{Bool}| & \stackrel{\text{def}}{=} & 1+1 \\
|A\times B| & \stackrel{\text{def}}{=} & |A|\times|B| \\
|A\to B| & \stackrel{\text{def}}{=} & |A|\to|B|
\end{array}
$$

Figure 3.5: Logical Types

$\boxed{c : A \sqsubseteq B}$

$$\frac{A \in \{1, ?\}}{\mathrm{id}(A) : A \sqsubseteq A} \qquad \frac{c : A' \sqsubseteq A'' \qquad d : A \sqsubseteq A'}{c \circ d : A \sqsubseteq A''} \qquad \mathrm{tag}(A) : G \sqsubseteq ?$$

$$\frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \times d : A_1 \times B_1 \sqsubseteq A_2 \times B_2} \qquad \frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \to d : A_1 \to B_1 \sqsubseteq A_2 \to B_2}$$

Figure 3.6: Term Assignment for Type Precision

Definition 12. An embedding projection pair of type $A \triangleleft B$ consists of

- An embedding $E_e[\cdot : A] : B$
- A projection $E_p[\cdot : B] : A$

Satisfying two properties:

- Retraction: $[\cdot] \approx^{\log} E_p[E_e[\cdot]]$
- Projection: $[\cdot] \sqsubseteq^{\text{ctx}} E_e[E_p[\cdot]]$

We want to define an ep pair for every type precision fact $A \sqsubseteq B$, and this is naturally structured as a recursion over the proof itself. To help make this explicit, we define a syntax for type precision proofs in Figure 3.6. The reflexivity rule is written as an identity and transitivity is written as a composition, which are suggestive of their semantics. Next, there is one difference with the type precision in Figure 2.8: the $A \sqsubseteq ?$ rule is restricted to have A be a tag type. This difference is motivated by the semantics we are about to present, but first note that it does not affect what is provable in the system:

Lemma 13. *For any gradual type A, there is a derivation $\mathrm{top}(A) : A \sqsubseteq ?$.*

Proof.

$$
\begin{array}{rcl}
\mathrm{top}(?) & \stackrel{\text{def}}{=} & \mathrm{id}(?) \\
\mathrm{top}(\texttt{Bool}) & \stackrel{\text{def}}{=} & \mathrm{tag}(\texttt{Bool}) \circ \mathrm{id}(?) \\
\mathrm{top}(A\times B) & \stackrel{\text{def}}{=} & \mathrm{tag}(?\times ?) \circ (\mathrm{top}(A)\times \mathrm{top}(B)) \\
\mathrm{top}(A\to B) & \stackrel{\text{def}}{=} & \mathrm{tag}(?\to ?) \circ (\mathrm{top}(A)\to \mathrm{top}(B))
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
m \in \{e, p\} & & & & & \\
\overline{e} & \stackrel{\text{def}}{=} & p & E_{e,c\circ d} & \stackrel{\text{def}}{=} & E_{e,c}[E_{e,d}] \\
\overline{p} & \stackrel{\text{def}}{=} & e & E_{p,c\circ d} & \stackrel{\text{def}}{=} & E_{p,d}[E_{p,c}] \\
E_{e,\mathrm{id}(A)} & \stackrel{\text{def}}{=} & [\cdot] & E_{m,c\times c'} & \stackrel{\text{def}}{=} & E_{m,c} \times E_{m,c'} \\
E_{e,\mathrm{tag}(G)} & \stackrel{\text{def}}{=} & \mathrm{roll}_{[\![?]\!]}\, \mathrm{inj}_G\, [\cdot] & E_{m,c\to c'} & \stackrel{\text{def}}{=} & E_{\overline{m},c} \to E_{m,c'} \\
E_{p,\mathrm{tag}(G)} & \stackrel{\text{def}}{=} & \mathrm{case}\ \mathrm{unroll}\ [\cdot]\ \mathrm{of}\ \mathrm{inj}_G\, x.\, x \mid \mathrm{else}.\, \mho &&&
\end{array}
$$

$$
\begin{array}{rcl}
E \times E' & \stackrel{\text{def}}{=} & \mathrm{let}\, \langle x, x' \rangle = [\cdot]\, \mathrm{in}\, \langle E[x], E'[x'] \rangle \\
E \to E' & \stackrel{\text{def}}{=} & \mathrm{let}\, x_f = [\cdot]\, \mathrm{in}\, \lambda x_a.\, E'[x_f\, (E[x_a])]
\end{array}
$$

Figure 3.7: Type Precision EP Pair Translation

$\boxed{c : A \sqsubseteq B}$

$$
\frac{A \in \{\mathtt{Bool}, ?\}}{\mathrm{id}(A) : A \sqsubseteq A}
\qquad
\frac{A \neq ? \qquad c : A \sqsubseteq \lfloor A \rfloor \qquad \overline{\mathrm{tag}(\lfloor A \rfloor) : \lfloor A \rfloor \sqsubseteq ?}}{\mathrm{tag}(\lfloor A \rfloor) \circ c : A \sqsubseteq ?}
$$

$$
\frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \times d : A_1 \times B_1 \sqsubseteq A_2 \times B_2}
\qquad
\frac{c : A_1 \sqsubseteq A_2 \qquad d : B_1 \sqsubseteq B_2}{c \to d : A_1 \to B_1 \sqsubseteq A_2 \to B_2}
$$

Figure 3.8: Canonical Forms for Type Precision Proofs

□

Next, we define the semantics of type precision proofs as embedding projection pairs in Figure 3.7. We defer the proof of the retraction and projection properties to §3.4.2.

Theorem 14. *For any derivation $c : A \sqsubseteq B$, $(E_{e,c}, E_{p,c}) : |A| \lhd |B|$.*

In particular, for every type $|A|$, there is an embedding projection pair $(E_{e,top(A)}, E_{p,top(A)}) : |A| \lhd |?|$.

Proof. Proven in §3.4. □

This establishes our semantics of types as coming with an ep pair from their logical type to the dynamic type.

Next we want to extend this to an interpretation of type precision $A \sqsubseteq B$ as saying there exists a unique ep pair from $|A|$ to $|B|$ that factorizes the ep pairs of A and B. By Lemma 14 we know that there is an ep pair from $|A|$ to $|B|$, but there are two related problems we need to address: it unique and does it factorize the ep pairs of A and B? As it stands, there are multiple proofs of any given fact $A \sqsubseteq B$, for instance we can prove $\mathtt{Int} \to \mathtt{Int} \sqsubseteq ?$ as $(\mathrm{tag}(\mathtt{Int}) \to \mathrm{tag}(\mathtt{Int})) \circ \mathrm{tag}(\to)$ or as

$$
\begin{array}{rcl}
\hat{\mathrm{id}}(?) & \stackrel{\text{def}}{=} & \mathrm{id}(?) \\
\hat{\mathrm{id}}(\mathtt{Bool}) & \stackrel{\text{def}}{=} & \mathrm{id}(\mathtt{Bool}) \\
\hat{\mathrm{id}}(A_1 \times A_2) & \stackrel{\text{def}}{=} & \hat{\mathrm{id}}(A_1) \times \hat{\mathrm{id}}(A_2) \\
\hat{\mathrm{id}}(A_1 \to A_2) & \stackrel{\text{def}}{=} & \hat{\mathrm{id}}(A_1) \to \hat{\mathrm{id}}(A_2)
\end{array}
$$

$$
\begin{array}{rcl}
(\mathrm{tag}(\lfloor A \rfloor) \circ c) \mathbin{\hat{\circ}} d & \stackrel{\text{def}}{=} & \mathrm{tag}(\lfloor A \rfloor) \circ (c \mathbin{\hat{\circ}} d) \\
(\mathrm{id}(A)) \mathbin{\hat{\circ}} d & \stackrel{\text{def}}{=} & d \\
(c \times d) \mathbin{\hat{\circ}} (c' \times d') & \stackrel{\text{def}}{=} & (c \mathbin{\hat{\circ}} c') \times (d \mathbin{\hat{\circ}} d') \\
(c \to d) \mathbin{\hat{\circ}} (c' \to d') & \stackrel{\text{def}}{=} & (c \mathbin{\hat{\circ}} c') \to (d \mathbin{\hat{\circ}} d')
\end{array}
$$

Figure 3.9: Type Precision Admissible Proof Terms

$(\mathrm{tag}(\mathtt{Int}) \to \mathrm{id}(\mathtt{Int})) \circ (\mathrm{id}(?) \to \mathrm{tag}(\mathtt{Int})) \circ \mathrm{tag}(\to)$, and on and on. These will all be given *a priori* distinct interpretations as ep pairs from $|\mathtt{Int} \to \mathtt{Int}|$ to $|?|$. This means that if we define a semantics of our proofs there is the issue of *coherence*: are different proofs interpreted as different embedding-projection pairs? We will show later that this is not the case: all proofs of the same type precision theorem construct *contextually equivalent* embedding-projection pairs. But for now, to give a single precise semantics we define *canonical forms* of proofs in Figure 3.8, and later show that the semantics of any term is equal to that of the canonical form. To do this we add three restrictions: (1) identity terms are restricted to base types $\mathtt{Bool}, ?$ (2) composition is restricted to composition with $\mathrm{tag}(G)$ and (3) $\mathrm{tag}(G)$ only appears in a composition. These restrictions do not affect provability because (1) all other identities can be expanded using the congruence rules (2) composition can all be associated away or pushed into congruence rules and (3) $\mathrm{tag}(G)$ is equivalent to $\mathrm{tag}(G) \circ \mathrm{id}(G)$.

Lemma 15. *Every type precision proof has a canonical form.*

Proof. Interpret $\mathrm{id}(A)$ as $\hat{\mathrm{id}}(A)$ and $c \circ d$ as $c \mathbin{\hat{\circ}} d$ using the definitions in Figure 3.9. □

Then our coherence theorem says that semantics is invariant under this canonicalization process.

Lemma 16. *If $c : A \sqsubseteq B$, and $Can(c) : A \sqsubseteq B$ is the canonicalized proof of c, then $E_{e,c} \approx^{log} E_{e,Can(c)}$ and $E_{p,c} \approx^{log} E_{p,Can(c)}$.*

As a consequence we have that type precision implies factorization of ep pairs to the dynamic type, and in fact more generally we have this for any $A \sqsubseteq B \sqsubseteq C$ that the ep pairs factorize:

Lemma 17. *If $b : A \sqsubseteq B$ and $c : B \sqsubseteq C$ and $d : A \sqsubseteq C$, then*

$$E_{e,d} \approx^{log} E_{e,c}[E_{e,b}]$$

$$E_{p,d} \approx^{log} E_{p,b}[E_{p,c}]$$

Proof. By Lemma 16,

$$E_{e,d} \approx^{\log} E_{e,c \,\hat{\circ}\, b} \approx^{\log} E_{e,c}[E_{e,b}]$$

And dually for the projections. □

Furthermore our canonicalized system gives us an easy way to formulate the decision procedure for type precision.

Theorem 18. *Given A, B it is decidable whether or not $A \sqsubseteq B$.*

Proof. We write a simple partial function from pairs of gradual types to proofs of ordering.

$$\mathrm{ord}(A, B) : A \sqsubseteq B$$

$$\begin{aligned}
\mathrm{ord}(A, ?) &= \mathrm{top}(A) \\
\mathrm{ord}(\texttt{Bool}, \texttt{Bool}) &= \texttt{Bool} \\
\mathrm{ord}(A \to B, A' \to B') &= \mathrm{ord}(A, A') \to \mathrm{ord}(B, B') \\
\mathrm{ord}(A \times B, A' \times B') &= \mathrm{ord}(A, A') \times \mathrm{ord}(B, B')
\end{aligned}$$

□

3.2.2 *Translations*

Next we give two translations from the cast calculus to the metalanguage that differ only in their treatment of casts. Since the cast calculus and metalanguage share so much of their syntax, most of the translation changes nothing, only the parts that are truly components of gradual typing need much translation.

Next, we define two elaborations of the cast calculus into the metalanguage in Figure 3.10. These translations are written $[\![\cdot]\!]^{ep}$ (the ep pair translation) and $[\![\cdot]\!]^{dir}$ (the direct translation). The two translations are mostly the same, and we use $[\![\cdot]\!]$ to stand for either of them. They differ only in their treatment of casts: $[\![\cdot]\!]^{ep}$ uses the embedding-projection pair of casts, while $[\![\cdot]\!]^{dir}$ provides an implementation that is more directly operationally analogous to the cast calculus. Both translations are type preserving in that if $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ then $x_1 : |A_1|, \ldots, x_n : |A_n| \vdash [\![t]\!] : |A|$.

The direct cast translation $E_{\langle B \Leftarrow A \rangle}$ of the appropriate type, is defined in Figure 3.11. Each case of the definition corresponds to one or more rules of the operational semantics. The product, sum, and function rules use the definitions of functorial actions of their types from Figure 3.7. We separate them because we will use the functoriality property in several definitions, theorems, and proofs later.

$\boxed{[\![t]\!]}$ where if $x_1 : A_1, \ldots, x_n : A_n \vdash t : A$ then $x_1 : [\![A_1]\!], \ldots, x_n : [\![A_n]\!] \vdash [\![t]\!] : [\![A]\!]$

$$
\begin{array}{rcl}
[\![\langle B \Leftarrow A\rangle t]\!]^{ep} & \stackrel{\text{def}}{=} & E_{p,\text{ord}(A,B)}[E_{e,\text{ord}(A,B)}[\![t]\!]^{ep}] \\
[\![\langle B \Leftarrow A\rangle t]\!]^{dir} & \stackrel{\text{def}}{=} & E_{\langle B \Leftarrow A\rangle}[[\![t]\!]^{dir}] \\
[\![x]\!] & \stackrel{\text{def}}{=} & x \\
[\![\langle t_1, t_2\rangle]\!] & \stackrel{\text{def}}{=} & \langle [\![t_1]\!] \, , [\![t_2]\!] \, \rangle \\
[\![\text{let } \langle x, y\rangle = t \text{ in } s]\!] & \stackrel{\text{def}}{=} & \text{let } \langle x, y\rangle = [\![t]\!] \text{ in } [\![s]\!] \\
[\![\texttt{true}]\!] & \stackrel{\text{def}}{=} & \text{inj}_1 \, \langle\rangle \\
[\![\texttt{false}]\!] & \stackrel{\text{def}}{=} & \text{inj}_2 \, \langle\rangle \\
[\![\texttt{if } t \texttt{ then } s \texttt{ else } s']\!] & \stackrel{\text{def}}{=} & \text{case } [\![t]\!] \text{ of } \text{inj}_1 \_. [\![s]\!] \mid \text{inj}_2 \_. [\![s']\!] \\
[\![\lambda(x : A).\, t]\!] & \stackrel{\text{def}}{=} & \lambda(x : [\![A]\!]).\, [\![t]\!] \\
[\![t\, s]\!] & \stackrel{\text{def}}{=} & [\![t]\!] \; [\![s]\!]
\end{array}
$$

Figure 3.10: Term Translation

$\boxed{E_{\langle B \Leftarrow A\rangle}}$ where $x : [\![A]\!] \vdash E_{\langle B \Leftarrow A\rangle}[x] : [\![B]\!]$

$$
\begin{array}{rcll}
E_{\langle ? \Leftarrow ?\rangle} & \stackrel{\text{def}}{=} & [\cdot] & \\
E_{\langle A_2 \times B_2 \Leftarrow A_1 \times B_1\rangle} & \stackrel{\text{def}}{=} & E_{\langle A_2 \Leftarrow A_1\rangle} \times E_{\langle B_2 \Leftarrow B_1\rangle} & \\
E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle} & \stackrel{\text{def}}{=} & E_{\langle A_1 \Leftarrow A_2\rangle} \to E_{\langle B_2 \Leftarrow B_1\rangle} & \\
E_{\langle ? \Leftarrow G\rangle} & \stackrel{\text{def}}{=} & \text{roll}_{[\![?]\!]} \, \text{inj}_G \, [\cdot] & \\
E_{\langle G \Leftarrow ?\rangle} & \stackrel{\text{def}}{=} & \text{case } (\text{unroll } [\cdot]) \text{ of } \text{inj}_G \, x.\, x \mid \text{else}.\, \mho & \\
E_{\langle ? \Leftarrow A\rangle} & \stackrel{\text{def}}{=} & E_{\langle ? \Leftarrow \lfloor A \rfloor\rangle}[E_{\langle \lfloor A \rfloor \Leftarrow A\rangle}[\cdot]] & \text{if } A \neq ?, \lfloor A \rfloor \\
E_{\langle A \Leftarrow ?\rangle} & \stackrel{\text{def}}{=} & E_{\langle A \Leftarrow \lfloor A \rfloor\rangle}[E_{\langle \lfloor A \rfloor \Leftarrow ?\rangle}[\cdot]] & \text{if } A \neq ?, \lfloor A \rfloor \\
E_{\langle B \Leftarrow A\rangle} & \stackrel{\text{def}}{=} & \text{let } x = [\cdot] \text{ in } \mho & \text{if } A, B \neq ? \text{ and } \lfloor A \rfloor \neq \lfloor B \rfloor
\end{array}
$$

Figure 3.11: Direct Cast Translation

3.2.3 *Direct Semantics is Adequate*

Next, we show that the direct semantics is *adequate* in that cast calculus programs terminate, diverge or error if and only if their direct semantics translations do. Later, we will show the ep pair semantics is also adequate by showing that it is equivalent to the direct semantics and then finally prove soundness of $\beta\eta$ equality and graduality.

We capture this relationship between the direct semantics and the cast calculus operational semantics in the following *forward* simulation theorem, which says that any reduction in the cast calculus corresponds to multiple steps in the target:

Lemma 19 (Translation Preserves Values, Evaluation Contexts)**.**

1. *For any value v, $[\![v]\!]^{dir}$ is a value.*
2. *For any evaluation context E, $[\![E]\!]^{dir}$ is an evaluation context.*

Lemma 20 (Simulation of Operational Semantics)**.** *If $t \mapsto t'$ then there exists s with $[\![t]\!]^{dir} \mapsto s \mapsto^* [\![t']\!]^{dir}$.*

Proof. By cases of $t \mapsto t'$. The non-cast cases are clear by lemma 19.

1. DynDyn

$$\begin{aligned} E_{\langle ?\Leftarrow ?\rangle}[[\![v]\!]^{dir}] &= \text{let } x = [\![v]\!]^{dir} \text{ in } x \\ &\mapsto [\![v]\!]^{dir} \end{aligned}$$

2. TagUp: Trivial because $[\![\langle ? \Leftarrow A\rangle v]\!]^{dir} = [\![\langle ? \Leftarrow \lfloor A\rfloor\rangle\langle \lfloor A\rfloor \Leftarrow A\rangle v]\!]^{dir}$.
3. TagDn: Trivial because $[\![\langle A \Leftarrow ?\rangle v]\!]^{dir} = [\![\langle A \Leftarrow \lfloor A\rfloor\rangle\langle \lfloor A\rfloor \Leftarrow ?\rangle v]\!]^{dir}$.
4. (TagMatch) Valid because

$$\begin{array}{ll} \text{case } (\text{unroll roll}_{[\![?]\!]^{dir}} \text{ inj}_G\, [\![v]\!]^{dir}) \text{ of} & \mapsto \text{case inj}_G\, [\![v]\!]^{dir} \text{ of} \\ \quad G.\, x & \qquad G.\, x \\ \quad \mid \text{else.}\, \mho & \qquad \mid \text{else.}\, \mho \\ & \mapsto [\![v]\!]^{dir} \end{array}$$

5. (TagMismatch) Valid because

$$\begin{array}{ll} \text{case unroll roll}_{[\![?]\!]^{dir}} \text{ inj}_{G'}\, [\![v]\!]^{dir} \text{ of} & \mapsto \text{case inj}_{G'}\, [\![v]\!]^{dir} \text{ of} \\ \quad G.\, x & \qquad G.\, x \\ \quad \mid \text{else.}\, \mho & \qquad \mid \text{else.}\, \mho \\ & \mapsto \mho \\ & = [\![\mho]\!]^{dir} \end{array}$$

6. (TagMismatch') Valid because

$$\text{let } x = [\![v]\!]^{dir} \text{ in } \mho \mapsto \mho$$

7. Pair Valid by

$$\begin{aligned}
&E_{\langle A_2 \times B_2 \Leftarrow A_1 \times B_1 \rangle}[\langle [\![v]\!]^{dir}, [\![v']\!]^{dir} \rangle] \\
&= \text{let } x = y \text{ in } \langle [\![v]\!]^{dir}, [\![v']\!]^{dir} \rangle \langle E_{\langle A_2 \Leftarrow A_1 \rangle}[x], E_{\langle B_2 \Leftarrow B_1 \rangle}[y] \rangle \\
&\mapsto \langle E_{\langle A_2 \Leftarrow A_1 \rangle}[[\![v]\!]^{dir}], E_{\langle B_2 \Leftarrow B_1 \rangle}[[\![v']\!]^{dir}] \rangle \\
&= [\![\langle \langle A_2 \Leftarrow A_1 \rangle v, \langle B_2 \Leftarrow B_1 \rangle v' \rangle]\!]^{dir}
\end{aligned}$$

8. (Fun) Valid because

$$\begin{aligned}
E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1 \rangle}[[\![v]\!]^{dir}] &\mapsto \text{let } x_f = [\![v]\!]^{dir} \text{ in} \\
&\mapsto \lambda(x_a : A_2).\, E_{\langle B_2 \Leftarrow B_1 \rangle}[[\![v]\!]^{dir}\, (E_{\langle A_1 \Leftarrow A_2 \rangle}[x_a])] \\
&= [\![\lambda(x_a : A_2).\, \langle B_2 \Leftarrow B_1 \rangle (v\, (\langle A_1 \Leftarrow A_2 \rangle x_a))]\!]^{dir}
\end{aligned}$$

□

To lift theorems for the gradual language from the typed language, we need to establish an *adequacy* theorem, which says that the translation's operational behavior determines the source. To do this we use the following backward simulation theorem.

Lemma 21 (Translation reflects Results)**.**

1. *If* $[\![t]\!]^{dir}$ *is a value,* $t \mapsto^* v$ *for some* v *with* $[\![t]\!]^{dir} = [\![v]\!]^{dir}$*.*
2. *If* $[\![t]\!]^{dir} = \mho$*, then* $t \mapsto^* \mho$*.*

Proof. By induction on t. For the non-casts, follows by inductive hypothesis. For the casts, only two cases can be values:

1. $\langle ? \Leftarrow ? \rangle t$: if $[\![\langle ? \Leftarrow ? \rangle t]\!]^{dir} = [\![t]\!]^{dir}$ is a value then by inductive hypothesis, t is a value, so $\langle ? \Leftarrow ? \rangle v \mapsto v$.
2. $\langle ? \Leftarrow G \rangle t$: if $\text{roll}_{[\![?]\!]^{dir}}\, \text{inj}_G\, [\![t]\!]^{dir}$ is a value, then $[\![t]\!]^{dir}$ is a value so by inductive hypothesis $t \mapsto^* v$ so $\langle ? \Leftarrow G \rangle t \mapsto^* \langle ? \Leftarrow G \rangle v$.

For the error case, there is only one case where it is possible for $[\![t]\!]^{dir} = \mho$ without $t = \mho$:

1. For $\langle ? \Leftarrow ? \rangle s$, if $[\![\langle ? \Leftarrow ? \rangle]\!]^{dir} [\![s]\!]^{dir} = [\![s]\!]^{dir}$ is an error then clearly $[\![s]\!]^{dir} = \mho$ so by inductive hypothesis $s \mapsto^* \mho$ and because casts are strict,

$$\langle ? \Leftarrow ? \rangle s \mapsto^* s$$

□

Lemma 22 (Backward Simulation)**.** *If* $[\![t]\!]^{dir} \mapsto s$ *then there exists* s' *with* $t \mapsto s'$ *and* $s \mapsto^* [\![s']\!]^{dir}$*.*

Proof. By induction on t. We show two illustrative cases, the rest follow by the same reasoning.

1. $[\![\text{let}\,\langle x, y\rangle = t \,\text{in}\, s]\!]^{dir} = \text{let}\,\langle x, y\rangle = [\![t]\!]^{dir} \,\text{in}\, [\![s]\!]^{dir}$. If $[\![t]\!]^{dir}$ is not a value, then we use the inductive hypothesis. If $[\![t]\!]^{dir}$ is a value and t is then by Lemma 21 $t \mapsto^* v$ and then we can reduce the pattern-match in source and target.

2. $[\![\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle t]\!]^{dir} = (E_{\langle A_1 \Leftarrow A_2\rangle} \to E_{\langle B_2 \Leftarrow B_1\rangle})[[\![t]\!]^{dir}]$. If $[\![t]\!]^{dir}$ is not a value, we use the inductive hypothesis. Otherwise, if it is a value and t is not, we use Lemma 21 to get $\langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle t \mapsto^* \langle A_2 \to B_2 \Leftarrow A_1 \to B_1\rangle v$. Then we use the same argument as the proof of Lemma 20.

3. $[\![\langle ? \Leftarrow A\rangle t]\!]^{dir} = E_{\langle ? \Leftarrow \lfloor A \rfloor\rangle}[E_{\langle \lfloor A \rfloor \Leftarrow A\rangle}[[\![t]\!]^{dir}]]$ then we use the same argument as the case for $\langle \lfloor A \rfloor \Leftarrow A\rangle t$, e.g., the function case above.

□

Theorem 23 (Adequacy)**.**

1. $[\![t]\!]^{dir} \mapsto^* v$ *if and only if* $t \mapsto^* v$ *with* $[\![v]\!]^{dir} = v$.
2. $[\![t]\!]^{dir} \mapsto^* \mho$ *if and only if* $t \mapsto^* \mho$.
3. $[\![t]\!]^{dir}$ *diverges if and only if* t *diverges*

Proof. The forward direction for values and errors is given by forward simulation Lemma 20. The backward direction for values and errors is given by induction on $[\![t]\!]^{dir} \mapsto^* t'$, backward simulation Lemma 22 and reflection of results lemma 21.

If t diverges, then by the backward value and error cases, it follows that $[\![t]\!]^{dir}$ does not run to a value or error. By type safety of the typed language, $[\![t]\!]^{dir}$ diverges.

Finally, if $[\![t]\!]^{dir}$ diverges, we show that t diverges. If $[\![t]\!]^{dir} \mapsto s$, then by backward simulation, there exists s' with $[\![t]\!]^{dir} \mapsto s'$ and $s \mapsto^* [\![s']\!]^{dir}$. Since $[\![t]\!]^{dir} \mapsto^* [\![s']\!]^{dir}$, we know $[\![s']\!]^{dir}$ diverges, so by coinduction s' diverges and therefore t diverges. □

3.3 REASONING ABOUT EQUIVALENCE AND ERROR APPROXIMATION

Next we turn to the goal of proving soundness of $\beta\eta$ equality and graduality for our cast calculus. To do this, we will develop a compositional semantic notion of error approximation, defined by induction on types: a step-indexed logical relation. To simplify reasoning, on top of the "implementation" of the logical relation as a step-indexed relation, we prove many high-level lemmas so that all proofs in the next sections are performed relative to these lemmas, and none manipulate step indices directly. Using this notion, we will prove that our interpretation of type precision as pairs of casts indeed forms embedding-projection pairs, which will lead to a clean proof of graduality.

3.3.1 *Logical Relation*

Our syntactic notions of $\beta\eta$ equality and term precision are inductively defined on the structure of terms, so it is natural to attempt to prove our semantic soundness theorem by induction over the relatedness derivations. However, the semantic principles that we have in mind (equivalence and error approximation) are only defined for closed terms, so they cannot serve as a useful inductive hypothesis for a proof of soundness. Logical relations are a technique for extending properties of closed terms compositionally to all terms of a language, and so we develop a logical relation to capture a general notion of error approximation for open terms of all types that is sound with respect to error approximation for closed terms.

Due to the non-well-founded nature of recursive types (and the dynamic type specifically), we develop a *step-indexed* logical relation following Ahmed [2]. We define our logical relation for error approximation in Figure 3.12.

Step-indexed logical relations are often used to define a slightly different notion of approximation: $t \sqsubseteq t'$ if either both have the same behavior or t *diverges* more, i.e., is less well-defined. Because our notion of error approximation is not this standard notion of approximation, we do something slightly unusual, which is we define *two* logical relations $\sqsubseteq\prec, \sqsubseteq\succ$, but the complexity is justified by the need to establish transitivity of the logical relation, which we discuss later (Lemma 31). For a given natural number $i \in \mathbb{N}$ and type A, and *closed terms* t_1, t_2 of type A, $t_1 \sqsubseteq\prec^i_{t,A} t_2$ intuitively means that, if we only inspect t_1's behavior up to i uses of unroll $\cdot$, then it appears that t_1 error approximates t_2. Less constructively, it means that we cannot show that t_1 does *not* error approximate t_2 when limited to i uses of unroll $\cdot$. However, even if we knew $t_1 \sqsubseteq\prec^i_{t,A} t_2$ for *every* $i \in \mathbb{N}$, it still might be the case that t_1 diverges, since no finite number of unrolling can ever exhaust t_1's behavior. So we also require that we know $t_1 \sqsubseteq\succ^i_{t,A} t_2$, which means that up to i uses of unroll on t_2, it appears that t_1 error approximates t_2.

The above intuition should help to understand the definition of error approximation for terms (i.e., the relations $\sqsubseteq\prec_t$ and $\sqsubseteq\succ_t$). The relation $t_1 \sqsubseteq\prec^i_{t,A} t_2$ is defined by inspection of t_1's behavior: it holds if t_1 is still running after $i+1$ unrolls; or if it steps to an error in fewer than i unrolls; or if it results in a value in fewer than i unrolls and also t_2 runs to a value and those values are related for the remaining steps. The definition of $t_1 \sqsubseteq\succ^i_{t,A} t_2$ is defined by inspection of t_2's behavior: it holds if t_2 is still running after $i+1$ unrolls; or if t_2 steps to an error in fewer than i steps then t_1 errors as well; or if t_2 steps to a value, either t_1 errors or steps to a value related for the remaining steps.

While the relations and $\sqsubseteq\succ_t$ on terms are different, fortunately, the relations on values are essentially the same, so we abstract over

$$
\begin{array}{rcl}
\sqsubseteq\prec^i_{t,A}, \sqsubseteq\succ^i_{t,A} & \subseteq & \{t \mid \cdot \vdash t : A\}^2 \\
t_1 \sqsubseteq\prec^i_{t,A} t_2 & \stackrel{\text{def}}{=} & (\exists t_1'.\ t_1 \mapsto^{i+1} t_1') \\
& & \vee (\exists j \leq i.\ t_1 \mapsto^{j} \mho) \\
& & \vee (\exists j \leq i, v_1 \sqsubseteq\prec^{i-j}_{v,A} v_2.\ t_1 \mapsto^{j} v_1 \wedge t_2 \mapsto^{*} v_2) \\
t_1 \sqsubseteq\succ^i_{t,A} t_2 & \stackrel{\text{def}}{=} & (\exists t_2'.\ t_2 \mapsto^{i+1} t_2') \\
& & \vee (\exists j \leq i.\ t_2 \mapsto^{j} \mho \wedge t_1 \mapsto^{*} \mho) \\
& & \vee (\exists j \leq i, v_2.\ t_2 \mapsto^{j} v_2 \wedge \\
& & \quad (t_1 \mapsto^{*} \mho \vee \exists v_1.\ t_1 \mapsto^{*} v_1 \wedge v_1 \sqsubseteq\succ^{i-j}_{v,A} v_2)
\end{array}
$$

$$
\begin{array}{rcll}
\sqsubseteq\prec^i_{v,A}, \sqsubseteq\succ^i_{v,A} & \subseteq & \{v \mid \cdot \vdash v : A\}^2 & \text{where } \sqsubseteq\sim \in \{\sqsubseteq\prec^{\cdot}_{\cdot,\cdot}, \sqsubseteq\succ^{\cdot}_{\cdot,\cdot}\} \\
v_1 \sqsubseteq\sim^0_{v,\mu\alpha.A} v_2 & \stackrel{\text{def}}{=} & \top & \\
\text{roll}_{\mu\alpha.A}\, v_1 \sqsubseteq\sim^{i+1}_{v,\mu\alpha.A} \text{roll}_{\mu\alpha.A}\, v_2 & \stackrel{\text{def}}{=} & v_1 \sqsubseteq\sim^i_{v,A[\alpha\mapsto\mu\alpha.A]} v_2 & \\
\langle\rangle \sqsubseteq\sim^i_{v,1} \langle\rangle & \stackrel{\text{def}}{=} & \top & \\
\langle v_1, v_1'\rangle \sqsubseteq\sim^i_{v,A\times A'} \langle v_2, v_2'\rangle & \stackrel{\text{def}}{=} & v_1 \sqsubseteq\sim^i_{v,A} v_2 \wedge v_1' \sqsubseteq\sim^i_{v,A'} v_2' & \\
v_1 \sqsubseteq\sim^i_{v,A+B} v_2 & \stackrel{\text{def}}{=} & (\exists (v_1' \sqsubseteq\sim^i_{v,A} v_2') \wedge v_1 = \text{inj}_1\, v_1' \wedge v_2 = \text{inj}_1\, v_2') & \\
& & \vee (\exists (v_1' \sqsubseteq\sim^i_{v,B} v_2') \wedge v_1 = \text{inj}_2\, v_1' \wedge v_2 = \text{inj}_2\, v_2') & \\
v_1 \sqsubseteq\sim^i_{v,A\to B} v_2 & \stackrel{\text{def}}{=} & \forall j \leq i. \forall (v_1' \sqsubseteq\sim^j_{v,A} v_2').\ v_1\ v_1' \sqsubseteq\sim^i_{t,B} v_2\ v_2' &
\end{array}
$$

$$
\begin{array}{rcl}
\cdot \sqsubseteq\sim^i_{v,\cdot} \cdot & \stackrel{\text{def}}{=} & \top \\
\gamma_1, v_1/x \sqsubseteq\sim^i_{v,\Gamma,x:A} \gamma_2, v_2/x & \stackrel{\text{def}}{=} & \gamma_1 \sqsubseteq\sim^i_{v,\Gamma} \gamma_2 \wedge v_1 \sqsubseteq\sim^i_{v,A} v_2
\end{array}
$$

$$
\Gamma \vDash t_1 \sqsubseteq\sim t_2 : A \stackrel{\text{def}}{=} \forall i \in \mathbb{N}, (\gamma_1 \sqsubseteq\sim^i_{v,\Gamma} \gamma_2).\ t_1[\gamma_1] \sqsubseteq\sim^i_{t,A} t_2[\gamma_2]
$$

$$
\Gamma \vDash t_1 \sqsubseteq t_2 : A \stackrel{\text{def}}{=} \Gamma \vdash t_1 \sqsubseteq\prec t_2 : A \wedge \Gamma \vdash t_1 \sqsubseteq\succ t_2 : A
$$

Figure 3.12: $\lambda_{T,\mho}$ Error Approximation Logical Relation

$$\frac{}{\Gamma \vDash \mho \sqsubseteq \mho : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vDash x \sqsubseteq x : A}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : A \qquad \Gamma, x : A \vDash s_1 \sqsubseteq s_2 : B}{\Gamma \vDash \mathsf{let}\, x = t_1 \,\mathsf{in}\, s_1 \sqsubseteq \mathsf{let}\, x = t_2 \,\mathsf{in}\, s_2 : B}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : A[\mu\alpha.\, A/\alpha]}{\Gamma \vDash \mathsf{roll}_{\mu\alpha.A}\, t_1 \sqsubseteq \mathsf{roll}_{\mu\alpha.A}\, t_2 : \mu\alpha.\, A}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : \mu\alpha.\, A}{\Gamma \vDash \mathsf{unroll}\, t_1 \sqsubseteq \mathsf{unroll}\, t_2 : A[\mu\alpha.\, A/\alpha]} \qquad \frac{}{\Gamma \vDash \langle\rangle \sqsubseteq \langle\rangle : 1}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : A \qquad \Gamma \vDash s_1 \sqsubseteq s_2 : B}{\Gamma \vDash \langle t_1, s_1\rangle \sqsubseteq \langle t_2, s_2\rangle : A \times B}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vDash s_1 \sqsubseteq s_2 : B}{\Gamma \vDash \mathsf{let}\, \langle x, y\rangle = t_1 \,\mathsf{in}\, s_1 \sqsubseteq \mathsf{let}\, \langle x, y\rangle = t_2 \,\mathsf{in}\, s_2 : B}$$

$$\frac{\Gamma \vDash t_1 \sqsubseteq t_2 : A + A' \qquad \Gamma, x : A \vDash s_1 \sqsubseteq s_2 : B \qquad \Gamma, x' : A' \vDash s_1' \sqsubseteq s_2' : B}{\Gamma \vDash \mathsf{case}\, t_1 \,\mathsf{of}\; \mathsf{inj}_1\, x.\, s_1 \mid \mathsf{inj}_2\, x'.\, s_1' \sqsubseteq \mathsf{case}\, t_2 \,\mathsf{of}\; \mathsf{inj}_1\, x.\, s_2 \mid \mathsf{inj}_2\, x'.\, s_2' : B}$$

$$\frac{\Gamma, x : A \vDash t_1 \sqsubseteq t_2 : B}{\Gamma \vDash \lambda x.\, t_1 \sqsubseteq \lambda x.\, t_2 : A \to B} \qquad \frac{\Gamma \vDash t_1 \sqsubseteq t_1 : A \to B \qquad \Gamma \vDash s_1 \sqsubseteq s_2 : A}{\Gamma \vDash t_1\, s_1 \sqsubseteq t_2\, s_2 : B}$$

Figure 3.13: $\lambda_{T,\mho}$ Error Approximation Congruence Rules

the cases by having the symbol $\sqsubseteq\sim$ to range over either $\sqsubseteq\prec$ or $\sqsubseteq\succ$. For values of recursive type, if the step-index is 0, we consider them related, because otherwise we would need to perform an unroll to inspect them further. Otherwise, we decrement the index and check if they are related. Decrementing the index here is exactly what makes the definition of the relation well-founded. For the standard types, the value relation definition is indeed standard: pairs are related when the two sides are related, sums must be the same case and functions must be related when applied to any related values in the future (i.e., when we may have exhausted some of the available steps).

Finally, we extend these relations to *open* terms in the standard way: we define substitutions to be related point-wise (similar to products) and then say that $\Gamma \vDash t_1 \sqsubseteq\sim t_2 : A$ holds if for every pair of substitutions γ_1, γ_2 related for i steps, the terms after substitution, written $t_1[\gamma_1]$ and $t_2[\gamma_2]$, are related for i steps. Then our resulting relation $\Gamma \vDash t_1 \sqsubseteq t_2$ is defined to hold when t_1 error approximates t_2 up to divergence of t_1 ($\sqsubseteq\prec$), *and* up to divergence of t_2 ($\sqsubseteq\succ$).

We need the following standard lemmas.

Lemma 24 (Downward Closure). *If $j \leq i$ then*

1. *If $t_1 \sqsubseteq{\sim}^{i}_{t,A} t_2$ then $t_1 \sqsubseteq{\sim}^{j}_{t,A} t_2$*
2. *If $v_1 \sqsubseteq{\sim}^{i}_{v,A} v_2$ then $v_1 \sqsubseteq{\sim}^{j}_{v,A} v_2$.*

Proof. By lexicographic induction on the pair (i, A). □

Lemma 25 (Anti-Reduction). *This theorem is different for the two relations as we allow arbitrary steps on the "divergence greater-than" side.*

1. *If $t_1 \sqsubseteq{\prec}^{i}_{t,A} t_2$ and $t_1' \mapsto^{j} t_1$ and $t_2' \mapsto^{*} t_2$ then $t_1' \sqsubseteq{\prec}^{i+j}_{t,A} t_2'$.*
2. *If $t_1 \sqsubseteq{\succ}^{i}_{t,A} t_2$ and $t_2' \mapsto^{j} t_2$ and $t_1' \mapsto^{*} t_1$, then $t_1' \sqsubseteq{\succ}^{i+j}_{t,A} t_2'$.*

A simple corollary that applies in common cases to both relations is that if $t_1 \sqsubseteq{\sim}^{i}_{t,A} t_2$ and $t_1' \mapsto^{0} t_1$ and $t_2' \mapsto^{0} t_2$, then $t_1' \sqsubseteq{\sim}^{i}_{t,A} t_2'$.

Proof. By direct inspection and downward closure (Lemma 24). □

Lemma 26 (Monadic Bind). *For any $i \in \mathbb{N}$, if for any $j \leq i$ and $v_1 \sqsubseteq{\sim}^{j}_{t,A} v_2$, we can show $E_1[v_1] \sqsubseteq{\sim}^{j}_{t,A} E_2[v_2]$ holds, then for any $t_1 \sqsubseteq{\sim}^{i}_{t,A} t_2$, it is the case that $E_1[t_1] \sqsubseteq{\sim}^{i}_{t,A} E_2[v_2]$.*

Proof. We consider the proof for $\sqsubseteq{\succ}^{i}_{t,A}$, the other is similar/easier. By case analysis of $t_1 \sqsubseteq{\succ}^{i}_{t,A} t_2$.

1. If t_2 takes $i+1$ steps, so does $E_2[t_2]$.
2. If $t_2 \mapsto^{j \leq i} \mho$ and $t_1 \mapsto^{*} \mho$, then first of all $E_2[t_2] \mapsto^{j} E_2[\mho] \mapsto \mho$. If $j+1 = i$, we are done. Otherwise $E_2[t_2] \mapsto^{j+1 \leq i} \mho$ and $E_1[t_1] \mapsto^{*} \mho$.
3. Assume there exist $j \leq i$, $v_1 \sqsubseteq{\succ}^{i-j}_{v,A} v_2$ and $t_2 \mapsto^{j} v_2$ and $t_1 \mapsto^{*} v_1$. Then by assumption, $E_1[v_1] \sqsubseteq{\succ}^{i-j}_{t,E_2} [v_2]$. Then by anti-reduction (Lemma 25), $E_1[t_1] \sqsubseteq{\succ}^{i}_{t,E_2} [t_2]$.

□

We then prove that our logical relation is sound for observational error approximation by showing that it is a congruence relation and showing that if we can prove error approximation up to divergence on the left *and* on the right, then we have true error approximation.

Lemma 27 (Congruence for Logical Relation). *All of the congruence rules in Figure 3.13 are valid.*

Proof. Each case is done by proving the implication for $\sqsubseteq{\prec}^{i}$ and $\sqsubseteq{\succ}^{i}$. Most cases follow by monadic bind (Lemma 26), downward closure (Lemma 24) and direct use of the inductive hypotheses. We show some illustrative cases.

1. Given $\gamma_1 \sqsubseteq\sim^i_{t,\Gamma} \gamma_2$, we need to show $\lambda(x : A).\, t_1[\gamma_1] \sqsubseteq\sim^i_{t,A\to B} \lambda(x : A).\, t_2[\gamma_2]$. Since they are values, we show they are related values. Given any $v_1 \sqsubseteq\sim^j_{v,A} v_2$ with $j \leq i$, each side β reduces in 0 unroll steps so it is sufficient to show

   $$t_1[\gamma_1, v_1'/x] \sqsubseteq\prec^j_{t,B} t_2[\gamma_2, v_2'/x]$$

   Which follows by inductive hypothesis and downward-closure and the substitution relation.

□

Theorem 28 (Logical Relation implies Error Approximation). *If* $\cdot \models t_1 \sqsubseteq t_2 : A$, *then* $t_1 \sqsubseteq_{err} t_2$

Proof. If $\cdot \models t_1 \sqsubseteq t_2 : A$, then since for any i, $\cdot \sqsubseteq\sim^i_{v,\cdot} \cdot$, we have $t_1 \sqsubseteq\sim^i_{v,A} t_2$ for any i (since $t_1[\cdot] = t_1$ and $t_2[\cdot] = t_2$).

We do a case analysis of t_1's behavior.

1. If $t_1 \mapsto^i \mho$ we're done.
2. If $t_1 \mapsto^i v$, then because $t_1 \sqsubseteq\prec^i_{t,A} t_2$, we know $t_2 \mapsto^* v'$.
3. If t_1 diverges, then for any i, $t_1 \mapsto^i t_1'$ and since $t_1 \sqsubseteq\succ^i_{t,A} t_2$, also we must have $t_2 \mapsto^i t_2'$, so t_2 diverges.

□

As a corollary, we also get that if we have the ordering in both directions, then the programs have equivalent behavior.

Corollary 29. *If* $\cdot \models t_1 \sqsubseteq t_2 : A$ *and* $\cdot \models t_2 \sqsubseteq t_1 : A$, *then* $t_1 \simeq t_2$.

So all we need to do to establish the soundness of $\beta\eta$ equality is to prove the validity of the generating $\beta\eta$ equations.

3.3.2 *Approximation and Equivalence Lemmas*

The step-indexed logical relation is on the face of it quite complex, especially due to the splitting of error approximation into two step-indexed relations. However, we should view the step-indexed relation as an "implementation" of the high-level concept of error approximation, and we work as much as possible with the error approximation relation $\Gamma \models t_1 \sqsubseteq t_2 : A$. In order to do this we now prove some high-level lemmas, which are proven using the step-indexed relations, but allow us to develop conceptual proofs of the key theorems of the paper.

In Chapter 5, we take this approach to its logical conclusion by developing a more sophisticated logic of error approximation.

First, there is reflexivity, also known as the *fundamental lemma*, which is proved using the same congruence cases as the soundness theorem (theorem 28.) Note that by the definition of our logical relation, this is really a kind of *monotonicity* theorem for every term in the language, the first component of our graduality proof.

Corollary 30 (Reflexivity). *If* $\Gamma \vdash t : A$ *then* $\Gamma \vDash t \sqsubseteq t : A$

Proof. By induction on the typing derivation of t, in each case using the corresponding congruence rule from Lemma 27. □

It is crucial to reasoning about ep pairs is to use the *transitivity* property, which is notoriously tedious to prove for step-indexed logical relations. Transitivity is often established through indirect reasoning—e.g., by setting up a biorthogonal ($\top\top$-closed) logical relation so one can easily show it is complete with respect to observational equivalence, which in turn implies that it must be transitive since observational equivalence is easily proven transitive. One direct method for proving transitivity, originally presented in [2], is to observe that two terms are observationally equivalent when each divergence approximates the other, and then prove that divergence approximation is transitive. Because a conjunction of transitive relations is transitive, this proves transitivity of equivalence. We have a similar issue with error approximation: the naïve logical relation for error approximation is not clearly transitive. Inspired by the case of observational equivalence, we similarly "split" our logical relation in two: $\sqsubseteq\prec, \sqsupseteq$. Unlike observational equivalence, the two relations are not the same. Instead, one $\sqsubseteq\prec$ is error approximation up to divergence on the *left* and the other $\sqsubseteq\succ$ is error approximation up to divergence on the *right*.

The proof works as follows: due to the function and open term cases, we cannot simply prove transitivity in the limit directly. Instead we get a kind of "asymmetric" transitivity: if $t_1 \sqsubseteq\prec^i_{t,A} t_2$ and *for any* $j \in \mathbb{N}$, $t_2 \sqsubseteq\prec^j_{t,A} t_3$, then we know $t_1 \sqsubseteq\prec^i_{t,A} t_3$. We abbreviate the $\forall j$ part as $t_2 \sqsubseteq\prec^\omega_{t,A} t_3$ in what follows. The key to the proof is in the function and open terms cases, which rely on reflexivity, corollary 30, as in Ahmed [2]. Reflexivity says that when we have $v_1 \sqsubseteq\prec^i_{v,A} v_2$ then we also have $v_2 \sqsubseteq\prec^\omega_{v,A} v_2$, which allows us to use the inductive hypothesis.

Lemma 31 (Transitivity for Closed Terms/Values). *The following are true for any* A.

1. *If* $t_1 \sqsubseteq\prec^i_{t,A} t_2$ *and* $t_2 \sqsubseteq\prec^\omega_{t,A} t_3$ *then* $t_1 \sqsubseteq\prec^i_{t,A} t_3$.
2. *If* $v_1 \sqsubseteq\prec^i_{t,A} v_2$ *and* $v_2 \sqsubseteq\prec^\omega_{t,A} v_3$ *then* $v_1 \sqsubseteq\prec^i_{t,A} v_3$.

Similarly,

1. *If* $t_1 \sqsubseteq\succ^\omega_{t,A} t_2$ *and* $t_2 \sqsubseteq\succ^i_{t,A} t_3$ *then* $t_1 \sqsubseteq\succ^i_{t,A} t_3$.
2. *If* $v_1 \sqsubseteq\succ^\omega_{t,A} v_2$ *and* $v_2 \sqsubseteq\succ^i_{t,A} v_3$ *then* $v_1 \sqsubseteq\succ^i_{t,A} v_3$.

Proof. We prove the $\sqsubseteq\prec^i_{t,A}$ and $\sqsubseteq\prec^i_{v,A}$ mutually by induction on (i, A). The other logical relation is similar. Most value cases are simple uses of the inductive hypotheses.

1. (Terms) By case analysis of $t_1 \sqsubseteq\prec^i_{t,A} t_2$.

a) If $t_1 \mapsto^{i+1} t_1'$ or $t_1 \mapsto^{j \leq i} \mathcal{U}$, we have the result.

b) Let $j \leq i$, $k \in \mathbb{N}$ and $(v_1 \sqsubseteq \prec^{i}_{v,A} v_2)$ with $t_1 \mapsto^{j} v_1$ and $t_2 \mapsto^{k} v_2$. By inductive hypothesis for values, it is sufficient to show that $t_3 \mapsto^{*} v_3$ and $v_2 \sqsubseteq \prec^{\omega}_{v,A} v_3$.

Since $t_2 \sqsubseteq \prec^{\omega}_{t,A} t_3$, in particular we know $t_2 \sqsubseteq \prec^{k+l}_{t,A} t_3$ for every $l \in \mathbb{N}$, so since $t_2 \mapsto^{k} v_2$, we know that $t_3 \mapsto^{*} v_3$ and $v_2 \sqsubseteq \prec^{l}_{v,A} v_3$, for every l, i.e., $v_2 \sqsubseteq \prec^{\omega}_{v,A} v_3$.

2. (Function values) Suppose $v_1 \sqsubseteq \prec^{i}_{v,A\to B} v_2$ and $v_2 \sqsubseteq \prec^{\omega}_{v,A\to B} v_3$. Then, let $j \leq i$ and $v_1' \sqsubseteq \prec^{j}_{v,A} v_2'$. We need to show $v_1\ v_1' \sqsubseteq \prec^{j}_{t,B} v_3\ v_2'$. By inductive hypothesis, it is sufficient to show $v_1\ v_1' \sqsubseteq \prec^{j}_{t,B} v_2\ v_2'$ and $v_2\ v_2' \sqsubseteq \prec^{\omega}_{t,B} v_3\ v_2'$.

   The former is clear. The latter follows by the congruence rule for application Lemma 27 and reflexivity corollary 30 on v_2': since $\cdot \vdash v_2' : A$, we have $v_2' \sqsubseteq \prec^{\omega}_{v,A} v_2'$.

□

Lemma 32 (Transitivity). *If* $\Gamma \vdash t_1 \sqsubseteq t_2 : A$ *and* $\Gamma \vdash t_2 \sqsubseteq t_3 : A$ *then* $\Gamma \vdash t_1 \sqsubseteq t_3 : A$.

Proof. The argument is essentially the same as the function value case, invoking the fundamental property corollary 30 for each component of the substitutions and transitivity for the closed relation Lemma 31 □

Next, we want to extract approximation and equivalence principles for *open* programs from syntactic operational properties of *closed* programs. First, obviously any operational reduction is a contextual equivalence, and the next lemma extends that to open programs. Note that we use $\sqsupseteq\sqsubseteq$ to mean approximation in both directions, i.e., equivalence:

Lemma 33 (Open β Reductions). *Given* $\Gamma \vdash t : A$, $\Gamma \vdash t' : A$, *if for every* $\gamma : \Gamma$, $t[\gamma] \mapsto^{*} t'[\gamma]$, *then* $\Gamma \vDash t \sqsupseteq\sqsubseteq t' : A$.

Proof. By reflexivity corollary 30 on t', γ and anti-reduction lemma 25. □

We call this open β reduction because we will use it to justify equivalences that look like an operational reduction, but have open values (i.e. including variables) rather than closed as in the operational semantics. For instance,

$$\text{let}\, x = y \,\text{in}\, t \sqsupseteq\sqsubseteq t[y/x]$$

and

$$\text{let}\, \langle x, y\rangle = \langle x', y'\rangle \,\text{in}\, t \sqsupseteq\sqsubseteq t[x'/x, y'/y]$$

Additionally, we need to establish the validity of η expansions.

$$\begin{array}{rcl} E[\mathsf{let}\, x = t\, \mathsf{in}\, s] & \sqsupseteq\sqsubseteq & \mathsf{let}\, x = t\, \mathsf{in}\, E[s] \\ E[\mathsf{let}\, \langle x, y\rangle = t\, \mathsf{in}\, s] & \sqsupseteq\sqsubseteq & \mathsf{let}\, \langle x, y\rangle = t\, \mathsf{in}\, E[s] \\ E[\mathsf{case}\, t\, \mathsf{of}\ \ \mathsf{inj}_1\, x.\, s \mid \mathsf{inj}_2\, x'.\, s'] & \sqsupseteq\sqsubseteq & \mathsf{case}\, t\, \mathsf{of}\ \ \mathsf{inj}_1\, x.\, E[s] \mid \mathsf{inj}_2\, x'.\, E[s'] \end{array}$$

Figure 3.14: Commuting Conversions

Lemma 34 (η Expansion)**.**

1. *For any* $\Gamma \vdash v : A \to B$, $v \sqsupseteq\sqsubseteq \lambda(x : A).\, v\, x$
2. *For any* $\Gamma, x : A + A', \Gamma' \vdash t : B$,

$$t \sqsupseteq\sqsubseteq \mathsf{case}\, x\, \mathsf{of}\ \ \mathsf{inj}_1\, y.\, t[\mathsf{inj}_1\, y'/x] \mid \mathsf{inj}_2\, y'.\, t[\mathsf{inj}_2\, y'/x]$$

3. *For any* $\Gamma, x : A \times A', \Gamma' \vdash t : B$,

$$t \sqsupseteq\sqsubseteq \mathsf{let}\, \langle y, y'\rangle = x\, \mathsf{in}\, t[\langle y, y'\rangle/x]$$

Proof. All are consequences of lemma 33. □

Next, with term constructors that involve continuations, we often need to rearrange the programs such as the "case-of-case" transformation. These are called commuting conversions and are presented in Figure 3.14.

Lemma 35 (Commuting Conversions)**.** *All of the commuting conversions in Figure 3.14 are equivalences.*

Proof. By monadic bind, anti-reduction and the reflexivity (lemmas 25 and 26 and corollary 30). □

Next, the following theorem is the main reason we so heavily use *evaluation contexts*. It is a kind of open version of the monadic bind lemma Lemma 26.

Lemma 36 (Evaluation contexts are linear)**.** *If* $\Gamma \vdash t : A$ *and* $\Gamma, x : A \vdash E[x] : B$*, then*

$$\mathsf{let}\, x = t\, \mathsf{in}\, E[x] \sqsupseteq\sqsubseteq E[t]$$

Proof. By a commuting conversion and an open β reduction, connected by transitivity lemmas 32, 33 and 35

$$\begin{array}{rcl} \mathsf{let}\, x = t\, \mathsf{in}\, E[x] & \sqsupseteq\sqsubseteq & E[\mathsf{let}\, x = t\, \mathsf{in}\, x] \\ & \sqsupseteq\sqsubseteq & E[t] \end{array}$$

□

As a simple example, consider the following standard equivalence of let and λ, which we will need later and prove using the above lemmas:

Lemma 37 (Let-λ Equivalence)**.** *For any* $\Gamma, x : A \vdash t : B$ *and* $\Gamma \vdash s : A$,

$$(\lambda(x : A).\, t)\, s \sqsupseteq\sqsubseteq \text{let } x = s \text{ in } t$$

Proof. First, we lift t using linearity of evaluation contexts, then an open β-reduction, linked by transitivity:

$$\begin{array}{lll} (\lambda(x : A).\, t)\, s \sqsupseteq\sqsubseteq \text{let } x = s \text{ in} & & \sqsupseteq\sqsubseteq \text{let } x = s \text{ in} \\ \qquad (\lambda(x : A).\, t)x & & \qquad t \end{array}$$

□

The concepts of pure and terminating terms are useful because when subterms are pure or terminating, they can be moved around to prove equivalences more easily.

Definition 38 (Pure, Terminating Terms)**.**

1. A term $\Gamma \vdash t : A$ is *terminating* if for any closing γ, either $t \mapsto^* \mho$ or $t \mapsto^* v$ for some v.
2. A term $\Gamma \vdash t : A$ is *pure* if for any closing γ, $t \mapsto^* v$ for some v.

The following terminology and proof are taken from Führmann [27].

Lemma 39 (Pure Terms are Thunkable)**.** *For any pure* $\Gamma \vdash t : A$,

$$\text{let } x = t \text{ in } \lambda(y : B).\, x \sqsupseteq\sqsubseteq \lambda(y : B).\, t$$

Proof. There are two cases $\sqsubseteq\prec, \sqsubseteq\succ$.

1. Let $\gamma_1 \sqsubseteq\prec^i_{v,\Gamma} \gamma_2$ and define $t_1 = t[\gamma_1]$ and $t_2 = t[\gamma_2]$. Then we know $t_1 \sqsubseteq\prec^i_{t,A} t_2$.
   a) If $t_1 \mapsto^{i+1}$ we're done.
   b) It's impossible that $t_1 \mapsto^* \mho$ because t is terminating.
   c) If $t_1 \mapsto^{j \le i} v_1$, then we know that $t_1 \mapsto^* v_2$ with $v_1 \sqsubseteq\prec^{i-j}_{v,A} v_2$. Next,

      $$\text{let } x = t_1 \text{ in } \lambda(y : B).\, x \mapsto^j \lambda(y : B).\, v_1$$

      Then it is sufficient to show $\lambda(y : B).\, \sqsubseteq\prec^{i-j}_{v,B\to A} \lambda(y : t_2).$, i.e. that for any $v_1' \sqsubseteq\prec^{k \le (i-j)}_{v,B} v_2'$ that

      $$(\lambda(y : B).\, v_1)\, v_1' \sqsubseteq\prec^k_{t,A} (\lambda(y : B).\, t_2)\, v_2'$$

The left side steps

$$(\lambda(y : B).\, v_1)\, v_1' \mapsto^0 v_1$$

And the right side steps

$$(\lambda(y : B).\, t_1)\, v_2' \mapsto^0 t_1 \mapsto^j v_2$$

And $v_1 \sqsubseteq \prec^k_{v,A} v_2$ by assumption above.

2. Let $\gamma_1 \sqsubseteq \succ^{i-j}_{\Gamma} \gamma_2$ and define $t_1 = t[\gamma_1]$ and $t_2 = t[\gamma_2]$. Then we know $t_1 \sqsubseteq \succ^{i}_{t,A} t_2$.

   Since t is terminating we know $t_1 \mapsto^* v_1$ and for some k, $t_2 \mapsto^k v_2$. Then we need to show And we need to show $\lambda(y : B).\, v_1 \sqsubseteq \succ^{i}_{t,B\to A} \lambda(y : B).\, t_2$. Given any $v_1' \sqsubseteq \succ^{k\le i}_{v,v_2'}$, we need to show

   $$(\lambda(y : B).\, v_1)\, v_1' \sqsubseteq \prec^k_{t,B} (\lambda(y : B).\, t_2)\, v_2'$$

   The β reduction takes 0 steps, then t_2 starts running. If $k > i$, we're done. Otherwise, $k \le i$ and we know $v_1 \sqsubseteq \succ^{i-k}_{v,A} v_2$ which is the needed result.

□

Lemma 40 (Pure Terms are Essentially Values)**.** *If $\Gamma \vdash t : A$ is a pure term, then for any $\Gamma, x : A \vdash s : B$*

$$\mathrm{let}\, x = t\, \mathrm{in}\, s \sqsupseteq\sqsubseteq s[t/x]$$

Proof. First, since by open β we have $(\lambda(y : 1).\, t\, \langle\rangle) \sqsupseteq\sqsubseteq t$, by congruence (lemma 27)

$$s[t/x] \sqsupseteq\sqsubseteq s[(\lambda(y : 1).\, t\, \langle\rangle)\, \langle\rangle / x]$$

And by reverse β reduction, this is further equivalent to

$$\mathrm{let}\, x_f = \lambda(y : 1).\, t\, \mathrm{in}$$
$$s[(x_f\, \langle\rangle)/x]$$

By thunkability of t and a commuting conversion this is equivalent to:

$$\mathrm{let}\, x = t\, \mathrm{in}$$
$$\mathrm{let}\, x_f = \lambda(y : 1).\, x\, \mathrm{in}$$
$$s[(x_f\, \langle\rangle)/x]$$

Which by β reduction at each x in x is:

$$\mathrm{let}\, x = t\, \mathrm{in}$$
$$\mathrm{let}\, x_f = \lambda(y : 1).\, x\, \mathrm{in}$$
$$s[x]$$

And a final β reduction eliminates the auxiliary x_f:

$$\begin{array}{l}\text{let } x = t \text{ in}\\ s[x]\end{array}$$

□

Also, since we consider all type errors to be equal, terminating terms can be reordered:

Lemma 41 (Terminating Terms Commute). *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash t' : A'$ *and* $\Gamma, x : A, y : A' \vdash s : B$*, then*

$$\text{let } x = t \text{ in let } x' = t' \text{ in } s \sqsupseteq\sqsubseteq \text{let } x' = t' \text{ in let } x = t \text{ in } s$$

Proof. By symmetry it is sufficient to prove one direction. Let $i \in \mathbb{N}$.

1. Let $\gamma_1 \sqsubseteq\prec i \Gamma \gamma_2$. We need to show

$$\begin{array}{ll}\text{let } x = t[\gamma_1] \text{ in} & \sqsubseteq\prec^i_{t,B} \text{ let } x' = t'[\gamma_2] \text{ in}\\ \text{let } x' = t'[\gamma_1] \text{ in} & \text{let } x = t\gamma_2 \text{ in}\\ s[\gamma_1] & s[\gamma_2]\end{array}$$

   Note that this is true if the left side diverges or errors, so this is true with no conditions on t, t'

   By corollary 30, we know $t[\gamma_1] \sqsubseteq\prec^i_{t,A} t[\gamma_2]$ and $t'[\gamma_1] \sqsubseteq\prec^i_{t,A'} t'[\gamma_2]$. We do a joint case analysis on these two facts.

   a) If $t[\gamma_1] \mapsto^{i+1}$, we're done.

   b) If $t[\gamma_1] \mapsto^{j \le i} \mho$, we're done.

   c) If $t[\gamma_1] \mapsto^{j \le i} v_1$, then also $t[\gamma_2] \mapsto^* v_2$.

      i. If $t'[\gamma_1] \mapsto^{(i-j)+1}$, we're done.

      ii. If $t'[\gamma_1] \mapsto^{k \le (i-j)} \mho$, we're done.

      iii. If $t'[\gamma_1] \mapsto^{k \le (i-j)} v_1'$, then $t'[\gamma_2] \mapsto^* v_2'$ with $v_1' \sqsubseteq\prec^{i-(j+k)}_{v,A'} v_2'$ and the result follows by corollary 30 for s because we know

$$s[\gamma_1, v_1/x, v_1'/x'] \sqsubseteq\prec^{i-(j+k)}_{t,A'} s[\gamma_2, v_2/x, v_2'/x']$$

2. Let $\gamma_1 \sqsubseteq\succ^i_{v,\Gamma} \gamma_2$. We need to show

$$\begin{array}{ll}\text{let } x = t[\gamma_1] \text{ in} & \sqsubseteq\succ^i_{t,B} \text{ let } x' = t'[\gamma_2] \text{ in}\\ \text{let } x' = t'[\gamma_1] \text{ in} & \text{let } x = t\gamma_2 \text{ in}\\ s[\gamma_1] & s[\gamma_2]\end{array}$$

   By corollary 30, we know $t[\gamma_1] \sqsubseteq\succ^i_{t,A} t[\gamma_2]$ and $t'[\gamma_1] \sqsubseteq\succ^i_{t,A'} t'[\gamma_2]$. We do a joint case analysis on these two facts.

a) If $t'[\gamma_2] \mapsto^{i+1}$, we're done.

b) If $t'[\gamma_2] \mapsto^{j \leq i} \mathcal{U}$ and $t'[\gamma_1] \mapsto^* \mathcal{U}$. In this case we know the right hand side errors, so we must show the left side errors. Since t is *terminating*, either $t[\gamma_1] \mapsto^* \mathcal{U}$ (done) or $t[\gamma_1] \mapsto^* v_1$. In the latter case we are also done because:

$$\begin{array}{ll} \text{let } x = t[\gamma_1] \text{ in } & \mapsto^* \text{let } x' = t'[\gamma_1] \text{ in } \mapsto^* \mathcal{U} \\ \text{let } x' = t'[\gamma_1] \text{ in } & s[\gamma_1, v/x] \\ s[\gamma_1] & \end{array}$$

c) If $t'[\gamma_2] \mapsto^{j \leq i} v_2'$ then either $t'[\gamma_1] \mapsto^* \mathcal{U}$ or $t'[\gamma_1] \mapsto^* v_1' \sqsubseteq \prec^{i-j}_{v,A'} v_2'$. Next, consider $t[\gamma_2]$.

   i. If $t[\gamma_2] \mapsto^{(i-j)+1}$ we're done.

   ii. If $t[\gamma_2] \mapsto^{k \leq (i-j)} \mathcal{U}$, then we know also that $t[\gamma_1] \mapsto^* \mathcal{U}$ so we're done.

   iii. If $t[\gamma_2] \mapsto^{k \leq (i-j)} v_2$, then either $t[\gamma_1] \mapsto^* \mathcal{U}$ or $t[\gamma_1] \mapsto^* v_1 \sqsubseteq \prec^{i-(j+k)}_{v,A} v_2$. If $t'[\gamma_1]$ or $t[\gamma_1]$ errors, we're done, otherwise both return values and the result follows by corollary 30 for s.

□

3.4 CASTS FROM EMBEDDING-PROJECTION PAIRS

In this section, we show how the casts in the cast calculus can be broken down into *embedding-projection* pairs. We will define embedding-projection pairs using our logical relation as the relevant ordering, and then prove the *coherence* theorem for the semantics of type precision derivations. Finally, we will show that the direct cast semantics is equivalent to the embedding-projection pair semantics, enabling our proofs of soundness of $\equiv$ and graduality.

3.4.1 *Embedding-Projection Pairs*

First, we define ep pairs with respect to *logical* approximation.

Definition 42 (EP Pair)**.** A embedding-projection pair $(E_e, E_p) : A \lhd B$ is a pair of an *embedding* $E_e[\cdot : A] : B$ and a *projection* $E_p[\cdot : B] : A$ satisfying

$$x : A \vDash x \sqsupseteq \sqsubseteq E_p[E_e[x]] : A \text{ RETRACTION}$$

$$y : B \vDash E_e[E_p[y]] \sqsubseteq y : B \text{ PROJECTION}$$

Next, we prove that in any embedding-projection pair that embeddings are pure (always produce a value) and that projections are

terminating (either error or produce a value). This agrees with our intuitions about casts: embedding a term in a (more) dynamic type should not produce any first-order errors, and for graduality to be true, adding a cast to a less dynamic type should not have any observable effects other than error. Paired with the lemmas we've proven about pure and terminating programs in the previous section, we will be able to prove theorems about ep pairs more easily.

In this language, purity of embeddings and termination of projections follows from the ep pair properties alone. However, the reasoning depends on the fact that the language only has divergence and error as effects. In Chapter 5 instead purity will be an additional axiom for embeddings, while termination will not be explicitly assumed for projections.

Lemma 43 (Embeddings are Pure). *If $E_e, E_p : A \lhd B$ is an embedding-projection pair then $x : A \vdash E_e[x] : B$ is pure.*

Proof. The ep pair property states that

$$x : A \vDash E_p[E_e[x]] \sqsupseteq\sqsubseteq x : A$$

Given any value $\cdot \vdash v : A$, by Lemma 30, we know

$$v \sqsubseteq\prec^0_{t,A} E_p[E_e[v]]$$

and since $v \mapsto^0 v$, this means there exists v' such that $E_p[E_e[v]] \mapsto^* v'$, and since E_p is an evaluation context, this means there must exist v'' with $E_p[E_e[v]] \mapsto^* E_p[v''] \mapsto^* v'$. □

Lemma 44 (Projections are Terminating). *If $E_e, E_p : A \lhd B$ is an embedding-projection pair then $y : B \vdash E_p[y] : A$ is terminating.*

Proof. The ep pair property states that

$$y : B \vDash E_e[E_p[y]] \sqsubseteq y : B$$

Given any $v : B$, by Lemma 30, we know

$$E_e[E_p[v]] \sqsubseteq\succ^0_{t,B} v$$

so therefore either $E_e[E_p[v]] \mapsto^* \mho$, which because E_e is pure means $E_p[v] \mapsto^* \mho$, or $E_e[E_p[v]] \mapsto^* v'$ which by strictness of evaluation contexts means $E_p[v] \mapsto^* v''$ for some v''. □

3.4.2 *Type Precision Semantics produce Coherent EP Pairs*

Next we turn to the task of proving that the embedding-projection pair semantics defined in §3.2.1 does in fact produce embedding-projection pairs, and that furthermore this semantics is coherent in that any two derivations produce equivalent ep pairs.

To start, we we need to prove that the identity pair is an ep pair, which is the semantics of reflexivity:

Lemma 45 (Identity EP Pair). *For any type A, $[\cdot], [\cdot] : A \lhd A$.*

Proof. Trivial. □

Second, if we compose the embeddings one way and projections the opposite way, the result is an ep pair, by congruence.

Lemma 46 (Composition of EP Pairs). *For any ep pairs* $E_{e,1}, E_{p,1} : A_1 \triangleleft A_2$ *and* $E_{e,1}, E_{e,2} : A_2 \triangleleft A_3$, $E_{e,2}[E_{e,2}], E_{p,1}[E_{p,2}] : A_1 \triangleleft A_3$.

Proof. By monadic bind Lemma 26. □

Next, we need to use *functoriality* of the actions of type constructors, meaning that the action of the type interacts well with identity and composition of evaluation contexts. First, we show that applying the functorial action of product and function types to identity functions produces identity functions.

Lemma 47 (Identity Expansion).

$$[\cdot] \times [\cdot] \sqsupseteq\sqsubseteq [\cdot] \quad \textit{and} \quad [\cdot] \to [\cdot] \sqsupseteq\sqsubseteq [\cdot]$$

Proof. All are instances of η expansion. □

In a call-by-value language, the composition is not preserved in general for the action of the product type, but composition of terminating programs is preserve because their order of evaluation is irrelevant. This is sufficient, since we are applying this construction to embeddings and projections, which by Lemmas 43 and 44 are terminating. Also notice that when composing using the functorial action of the function type $\to$, the composition flips on the domain side, because the function type is *contravariant* in its domain.

Lemma 48 (Functoriality for Terminating Programs). *The following equivalences are true for any well-typed,* terminating *evaluation contexts.*

$$\begin{array}{rcl} (E_2 \times E_2')[E_1 \times E_1'] & \sqsupseteq\sqsubseteq & (E_2[E_1]) \times (E_2'[E_1']) \\ (E_2 \to E_2')[E_1 \to E_1'] & \sqsupseteq\sqsubseteq & (E_1[E_2]) \to (E_2'[E_1']) \end{array}$$

Proof. 1. ($\to$) We need to show (after a commuting conversion)

$$\begin{array}{lcl} \text{let } x_f = [\cdot] \text{ in} & \sqsupseteq\sqsubseteq & \text{let } x_f = [\cdot] \text{ in} \\ \text{let } y_f = \lambda x_a.\, E_1'[x_f\, (E_1[x_a])] \text{ in} & & \lambda y_a.\, E_2'[E_1'[x_f\, (E_1[E_2[y_a]])]] \\ \lambda y_a.\, E_2'[y_f\, (E_2[y_a])] & & \end{array}$$

First, we substitute for y_f and then lift the argument $E_2[y_a]$ out and β reduce:

$$\begin{array}{ll}
\text{let } x_f = [\cdot] \text{ in} & \sqsupseteq\sqsubseteq \text{let } x_f = [\cdot] \text{ in} \\
\text{let } y_f = \lambda x_a.\, E_1'[x_f\ (E_1[x_a])] \text{ in} & \quad \lambda y_a.\, E_2'[(\lambda x_a.\, E_1'[x_f\ (E_1[x_a])])\ (E_2[y_a])] \\
\lambda y_a.\, E_2'[y_f\ (E_2[y_a])] & \\
& \sqsupseteq\sqsubseteq \text{let } x_f = [\cdot] \text{ in} \\
& \quad \lambda y_a.\, \text{let } x_a = E_2[y_a] \text{ in} \\
& \qquad E_2'[(\lambda x_a.\, E_1'[x_f\ (E_1[x_a])])\ x_a] \\
& \sqsupseteq\sqsubseteq \text{let } x_f = [\cdot] \text{ in} \\
& \quad \lambda y_a.\, \text{let } x_a = E_2[y_a] \text{ in} \\
& \qquad E_2'[E_1'[x_f\ (E_1[x_a])]] \\
& \sqsupseteq\sqsubseteq \text{let } x_f = [\cdot] \text{ in} \\
& \quad \lambda y_a.\, E_2'[E_1'[x_f\ (E_1[E_2[y_a]])]]
\end{array}$$

2. ($\times$) We need to show

$$\begin{array}{ll}
\text{let } \langle x, x' \rangle = x_p \text{ in} & \sqsupseteq\sqsubseteq \text{let } \langle x, x' \rangle = x_p \text{ in} \\
\text{let } \langle y, y' \rangle = \langle E_1[x], E_1'[x'] \rangle \text{ in} & \quad \langle E_2[E_1[x]], E_2' E_1'[x'] \rangle \\
\langle E_2[y], E_2'[y'] \rangle &
\end{array}$$

First, we make the evaluation order explicit, then re-order using the fact that terminating programs commute lemma 41.

$$
\begin{array}{ll}
\text{let } \langle x, x' \rangle = x_p \text{ in} & \sqsupseteq\sqsubseteq \text{let } \langle x, x' \rangle = x_p \text{ in} \\
\text{let } \langle y, y' \rangle = \langle E_1[x], E_1'[x'] \rangle \text{ in} & \quad \text{let } x_1 = E_1[x] \text{ in} \\
\langle E_2[y], E_2'[y'] \rangle & \quad \text{let } x_1' = E_1'[x'] \text{ in} \\
& \quad \text{let } \langle y, y' \rangle = \langle x_1, x_1' \rangle \text{ in} \\
& \quad \text{let } x_2 = E_2[y] \text{ in} \\
& \quad \text{let } x_2' = E_2'[y'] \text{ in} \\
& \quad \langle x_2, x_2' \rangle \\
& \sqsupseteq\sqsubseteq \text{let } \langle x, x' \rangle = x_p \text{ in} \\
& \quad \text{let } x_1 = E_1[x] \text{ in} \\
& \quad \text{let } x_1' = E_1'[x'] \text{ in} \\
& \quad \text{let } x_2 = E_2[x_1] \text{ in} \\
& \quad \text{let } x_2' = E_2'[x_1'] \text{ in} \\
& \quad \langle x_2, x_2' \rangle \\
& \sqsupseteq\sqsubseteq \text{let } \langle x, x' \rangle = x_p \text{ in} \\
& \quad \text{let } x_1 = E_1[x] \text{ in} \\
& \quad \text{let } x_2 = E_2[x_1] \text{ in} \\
& \quad \text{let } x_1' = E_1'[x'] \text{ in} \\
& \quad \text{let } x_2' = E_2'[x_1'] \text{ in} \\
& \quad \langle x_2, x_2' \rangle \\
& \sqsupseteq\sqsubseteq \text{let } \langle x, x' \rangle = x_p \text{ in} \\
& \quad \langle E_2[E_1[x]], E_2'[E_1'[x']] \rangle
\end{array}
$$

□

With these cases covered, we can show the casts given by type precision really are ep pairs.

Lemma 49 (Type Precision Derivations denote EP Pairs)**.**

Proof of Theorem 14. We seek to probe that for any derivation $c : A \sqsubseteq B$, that $E_{e,c}, E_{p,c} : [\![A]\!] \triangleleft [\![B]\!]$ are an ep pair. We proceed by induction on the derivation c.

1. (Identity) $E_{e,\mathrm{id}(A)}, E_{p,\mathrm{id}(A)} : [\![A]\!] \triangleleft [\![A]\!]$. This case is immediate by corollary 30.
2. (Composition) $\dfrac{E_{e,c}, E_{p,c} : [\![A_1]\!] \triangleleft [\![A_2]\!] \qquad E_{e,c'}, E_{p,c'} : [\![A_2]\!] \triangleleft [\![A_3]\!]}{E_{e,c'}[E_{e,c}], E_{p,c}[E_{p,c'}] : [\![A_1]\!] \triangleleft [\![A_3]\!]}$.
   We need to show the retract property:

$$x : [\![A_1]\!] \vDash E_{p,c}[E_{p,c'}[E_{e,c'}[E_{e,c}[x]]]] \sqsupseteq\sqsubseteq x : [\![A_1]\!]$$

and the projection property:

$$y : [\![A_3]\!] \vDash E_{e,c'}[E_{e,c}[E_{p,c}[E_{p,c'}[y]]]] \sqsubseteq y : [\![A_3]\!]$$

Both follow by congruence and the inductive hypothesis, we show the projection property:

$$\begin{aligned} E_{e,c'}[E_{e,c}[E_{p,c}[E_{p,c'}[y]]]] &\sqsubseteq E_{e,c'}[E_{p,c'}[y]] && \text{(inductive hyp, cong Lemma 27)} \\ &\sqsubseteq y && \text{(inductive hyp)} \end{aligned}$$

3. (Tag) $\dfrac{}{\begin{array}{l}\mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G\ [\cdot], \mathsf{case}\ \mathsf{unroll}\ [\cdot]\ \mathsf{of}\ \mho : [\![A]\!] \triangleleft [\![B]\!] \\ \qquad\qquad [\![G]\!].\ x \\ \qquad\quad \mid \mathsf{else}.\ x\end{array}}$.

The retraction case follows by β reduction

$$\begin{array}{l}\mathsf{case}\ \mathsf{unroll}\ \mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G\ x\ \mathsf{of}\ \sqsupseteq\sqsubseteq x \\ \quad \mathsf{inj}_G\ x_G.\ x_G \\ \quad \mid \mathsf{else}.\ \mho\end{array}$$

For the projection case, we need to show

$$\mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G \left(\begin{array}{l}\mathsf{case}\ \mathsf{unroll}\ y\ \mathsf{of} \\ \quad \mathsf{inj}_G\ x_G.\ x_G \\ \quad \mid \mathsf{else}.\ \mho\end{array}\right) \sqsubseteq y$$

First, on the left side, we do a commuting conversion (lemma 35) and then use linearity of evaluation contexts to reduce the cases to error:

$$\begin{aligned} \mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G \left(\begin{array}{l}\mathsf{case}\ \mathsf{unroll}\ y\ \mathsf{of} \\ \quad \mathsf{inj}_G\ x_G.\ x_G \\ \quad \mid \mathsf{else}.\ \mho\end{array}\right) &\sqsupseteq\sqsubseteq \begin{array}{l}\mathsf{case}\ \mathsf{unroll}\ y\ \mathsf{of} \\ \quad \mathsf{inj}_G\ x_G.\ \mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G\ x_G \\ \quad \mid \mathsf{else}.\ \mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G\ \mho\end{array} \\ &\sqsupseteq\sqsubseteq \begin{array}{l}\mathsf{case}\ \mathsf{unroll}\ y\ \mathsf{of} \\ \quad \mathsf{inj}_G\ x_G.\ \mathsf{roll}_{[\![?]\!]}\ \mathsf{inj}_G\ x_G \\ \quad \mid \mathsf{else}.\ \mho\end{array} \end{aligned}$$

Next, we η-expand the right hand side

$$
\begin{array}{l}
y \sqsupseteq\sqsubseteq \mathrm{roll}_{[\![?]\!]}\ \mathrm{unroll}\ y \\
\quad \sqsupseteq\sqsubseteq \mathrm{case}\ \mathrm{unroll}\ y\ \mathrm{of} \\
\qquad \mathrm{inj}_{\mathbb{B}}\ x_{\mathbb{B}}.\ \mathrm{roll}_{[\![?]\!]}\ \mathrm{inj}_{\mathbb{B}}\ x_{\mathbb{B}} \\
\qquad \mathrm{inj}_{+}\ x_{+}.\ \mathrm{roll}_{[\![?]\!]}\ \mathrm{inj}_{+}\ x_{+} \\
\qquad \mathrm{inj}_{\times}\ x_{\times}.\ \mathrm{roll}_{[\![?]\!]}\ \mathrm{inj}_{\times}\ x_{\times} \\
\qquad \mathrm{inj}_{\to}\ x_{\to}.\ \mathrm{roll}_{[\![?]\!]}\ \mathrm{inj}_{\to}\ x_{\to}
\end{array}
$$

The result follows by congruence because $\mho \sqsubseteq t$ for any t.

4. (Functions) $\dfrac{E_{e,c}, E_{p,c} : A_1 \lhd A_2 \qquad E_{e,c'}, E_{p,c'} : B_1 \lhd B_2}{E_{p,c} \to E_{e,c'}, E_{e,c} \to E_{p,c'} : [\![A_1]\!] \to [\![B_1]\!] \lhd [\![A_2]\!] \to [\![B_2]\!]}$
We prove the projection property, the retraction proof is similar. We want to show

$$y : [\![A_2]\!] \to [\![B_2]\!] \vDash (E_{e,c} \to E_{p,c'})[(E_{p,c} \to E_{e,c'})[y]] \sqsubseteq y : [\![A_2]\!] \to [\![B_2]\!]$$

Since embeddings and projections are terminating, we can apply functoriality Lemma 48 to show the left hand side is equivalent to

$$((E_{p,c}[E_{e,c}]) \to (E_{p,c'}[E_{e,c'}]))[y]$$

which by congruence and inductive hypothesis is $\sqsubseteq$:

$$([\cdot] \to [\cdot])[y]$$

which by identity extension lemma 47 is equivalent to y.

5. (Products) By the same argument as the function case.

□

Next, key to proving our coherence theorem is the fact that the admissible rules for reflexivity ($\hat{\mathrm{id}}(A)$) and composition $c \mathbin{\hat{\circ}} d$ defined in Figure 3.9 are equivalent to the identity and composition of evaluation contexts. First, we show, $\hat{\mathrm{id}}(A)$ is the identity by identity extension.

Lemma 50 (Reflexivity Proofs denote Identity)**.** *For every A, $E_{e,\hat{id}(A)} \sqsupseteq\sqsubseteq [\cdot]$ and $E_{p,\hat{id}(A)} \sqsupseteq\sqsubseteq [\cdot]$.*

Proof. By induction on A, using the identity extension lemma. □

Second, we prove our key *decomposition* theorem. While the composition theorem says that the composition of any two ep pairs is an ep pair, the *decomposition* theorem is really a theorem about the *coherence* of our type precision proofs. It says that given any ep pair given by $c : A_1 \sqsubseteq A_3$, if we can find a middle type A_2, then we can *decompose* c's ep pairs into a composition. This theorem is used extensively, especially in the proof of the gradual guarantee.

Lemma 51 (Decomposition of Upcasts, Downcasts)**.** *For any derivations* $c : A_1 \sqsubseteq A_2$ *and* $c' : A_2 \sqsubseteq A_3$*, the upcasts and downcasts given by their composition* $c' \mathbin{\hat{\circ}} c$ *are equivalent to the composition of their casts given by* c, c'*:*

$$x : [\![A_1]\!] \vDash E_{e,c' \mathbin{\hat{\circ}} c}[x] \sqsupseteq\sqsubseteq E_{e,c'}[E_{e,c}[x]] : [\![A_3]\!]$$
$$y : [\![A_3]\!] \vDash E_{p,c' \mathbin{\hat{\circ}} c}[y] \sqsupseteq\sqsubseteq E_{p,c}[E_{p,c'}[y]] : [\![A_1]\!]$$

Proof. By induction on the pair c, following the recursive definition of $c \mathbin{\hat{\circ}} c'$.

1. $(\mathrm{tag}(G) \circ c) \mathbin{\hat{\circ}} d \stackrel{\text{def}}{=} \mathrm{tag}(G) \circ (c \mathbin{\hat{\circ}} d)$. By inductive hypothesis and strict associativity of composition of evaluation contexts.

2. $\mathrm{id}(A_1) \mathbin{\hat{\circ}} d \stackrel{\text{def}}{=} d$ reflexivity.

3. $(c \times d) \mathbin{\hat{\circ}} (c' \times d') \stackrel{\text{def}}{=} (c \mathbin{\hat{\circ}} c') \times (d \mathbin{\hat{\circ}} d')$ By inductive hypothesis and functoriality lemma 48.

4. $(c \to d) \mathbin{\hat{\circ}} (c' \to d') \stackrel{\text{def}}{=} (c \mathbin{\hat{\circ}} c') \to (d \mathbin{\hat{\circ}} d')$ By inductive hypothesis and functoriality lemma 48.

□

The identity and decomposition lemmas are then enough to prove that the semantics of type precision proofs is coherent.

Proof of Theorem 16. We seek to prove that for any $c : A_1 \sqsubseteq A_2$, if $Can(c) : A_1 \sqsubseteq A_2$ is the canonicalized proof, then their corresponding embeddings and projections are equal. The proof follows by induction on c. If c is and identity, we use Lemma 50 and if c is a composition we use Lemma 51. □

Finally, now that we have established the meaning of type precision derivations and proven the decomposition theorem, we can dispense with direct manipulation of derivations. So we define the following notation for ep pairs that just uses the types:

Definition 52 (EP Pair Semantics)**.** Given $c : A \sqsubseteq B$, we define $E_{e,A,B} = E_{e,c}$, and $E_{p,A,B} = E_{p,c}$.

3.4.3 *Casts Factorize into EP Pairs*

Next, we prove the equivalence between the direct semantics and ep pair semantics.

First, when $A \sqsubseteq B$, the direct cast semantics for a cast from A to B is the equivalent to the embedding of A into B and vice-versa for the projection.

Lemma 53 (Upcasts and Downcasts are Casts)**.** *If* $A \sqsubseteq B$ *then* $E_{\langle B \Leftarrow A \rangle} \sqsupseteq\sqsubseteq E_{e,A,B}$ *and* $E_{\langle A \Leftarrow B \rangle} \sqsupseteq\sqsubseteq E_{p,A,B}$.

Proof. By induction following the recursive definition of $E_{\langle B \Leftarrow A \rangle}$

1. $E_{\langle ? \Leftarrow ? \rangle} \stackrel{\text{def}}{=} [\cdot]$ By reflexivity.

2. $E_{\langle A_2 \times B_2 \Leftarrow A_1 \times B_1 \rangle} \stackrel{\text{def}}{=} E_{\langle A_2 \Leftarrow A_1 \rangle} \times E_{\langle B_2 \Leftarrow B_1 \rangle}$ By inductive hypothesis and congruence.

3. $E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1 \rangle} \stackrel{\text{def}}{=} E_{\langle A_1 \Leftarrow A_2 \rangle} \to E_{\langle B_2 \Leftarrow B_1 \rangle}$ By inductive hypothesis and congruence.

4. $E_{\langle ? \Leftarrow G \rangle} \stackrel{\text{def}}{=} \text{roll}_{[\![?]\!]} \, \text{inj}_G \, [\cdot]$ By reflexivity

5. $E_{\langle G \Leftarrow ? \rangle} \stackrel{\text{def}}{=} \text{case unroll} \, [\cdot] \, \text{of inj}_G \, x. \, x \mid \text{else}. \, \mho$ By reflexivity.

6. $(A \neq ?, \lfloor A \rfloor) \quad E_{\langle ? \Leftarrow A \rangle} \stackrel{\text{def}}{=} E_{\langle ? \Leftarrow \lfloor A \rfloor \rangle}[E_{\langle \lfloor A \rfloor \Leftarrow A \rangle}[\cdot]]$ By inductive hypothesis and decomposition of ep pairs.

7. $(A \neq ?, \lfloor A \rfloor) \quad E_{\langle A \Leftarrow ? \rangle} \stackrel{\text{def}}{=} E_{\langle A \Leftarrow \lfloor A \rfloor \rangle}[E_{\langle \lfloor A \rfloor \Leftarrow ? \rangle}[\cdot]]$ By inductive hypothesis and decomposition of ep pairs.

8. $(A, B \neq ? \wedge \lfloor A \rfloor \neq \lfloor B \rfloor) \quad E_{\langle B \Leftarrow A \rangle} \stackrel{\text{def}}{=} \text{let } x = [\cdot] \text{ in } \mho$. Not possible that $A \sqsubseteq B$.

□

Next, we show that the "general" casts of the gradual language can be *factorized* into a composition of an upcast followed by a downcast. First, we show that factorizing through any type is equivalent to factorizing through the dynamic type, as a consequence of the *retraction* property of ep pairs.

Lemma 54 (Any Factorization is equivalent to Dynamic)**.** *For any A_1, A_2, A' with $A_1 \sqsubseteq A'$ and $A_2 \sqsubseteq A'$, $E_{p,A_2,?}[E_{e,A_1,?}] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A_1,A'}]$.*

Proof. By decomposition and the retraction property:

$$E_{p,A_2,?}[E_{e,A_1,?}] \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{p,A',?}[E_{e,A',?}[E_{e,A_1,?}]]] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A_1,A'}]$$

□

By transitivity of equivalence, this means that factorization through one B is as good as any other. So to prove that every cast factors as an upcast followed by a downcast, we can choose whatever middle type is most convenient. This lets us choose the simplest type possible in the proof. For instance, when factorizing a function cast $\langle A_2 \to B_2 \Leftarrow A_1 \to B_1 \rangle$, we can use the function tag type as the middle type $? \to ?$ and then the equivalence is a simple use of the inductive hypothesis and the functoriality principle.

Lemma 55 (Every Cast Factors as Upcast, Downcast). *For any A_1, A_2, A' with $A_1 \sqsubseteq A'$ and $A_2 \sqsubseteq A'$, the cast from A_1 to A_2 factorizes through A':*

$$x : [\![A]\!] \vDash E_{\langle A_2 \Leftarrow A_1 \rangle}[x] \sqsupseteq\sqsubseteq E_{p,A_2,A'}[E_{e,A_1,A'}[x]] : [\![A_2]\!]$$

Proof.

1. If $A_1 \sqsubseteq A_2$, then we choose $A' = A_2$ and we need to show that

   $$E_{\langle A_2 \Leftarrow A_1 \rangle} \sqsupseteq\sqsubseteq E_{p,A_2,A_2}[E_{e,A_1,A_2}]$$

   this follows by lemma 53 and lemma 50.

2. If $A_2 \sqsubseteq A_1$, we use a dual argument to the previous case. We choose $A' = A_1$ and we need to show that

   $$E_{\langle A_2 \Leftarrow A_1 \rangle} \sqsupseteq\sqsubseteq E_{p,A_1,A_2}[E_{e,A_1,A_1}]$$

   this follows by lemma 53 and lemma 50.

3. $E_{\langle A_2 \times B_2 \Leftarrow A_1 \to B_1 \rangle} \stackrel{\text{def}}{=} E_{\langle A_2 \Leftarrow A_1 \rangle} \to E_{\langle B_2 \Leftarrow B_1 \rangle}$ We choose $A' = ? \to ?$. By inductive hypothesis,

   $$E_{\langle A_1 \Leftarrow A_2 \rangle} \sqsupseteq\sqsubseteq E_{p,A_1,?}[E_{e,A_2,?}] \quad \text{and} \quad E_{\langle B_2 \Leftarrow B_1 \rangle} \sqsupseteq\sqsubseteq E_{p,B_2,?}[E_{e,B_1,?}]$$

   Then the result holds by functoriality:

   $$\begin{aligned} E_{\langle A_2 \to B_2 \Leftarrow A_1 \to B_1 \rangle} &= E_{\langle A_1 \Leftarrow A_2 \rangle} \to E_{\langle B_2 \Leftarrow B_1 \rangle} \\ &\sqsupseteq\sqsubseteq (E_{p,A_1,?}[E_{e,A_2,?}]) \to (E_{p,B_2,?}[E_{e,B_1,?}]) \\ &\sqsupseteq\sqsubseteq (E_{e,A_2,?} \to E_{p,B_2,?})[E_{p,A_1,?} \to E_{e,B_1,?}] \\ &= E_{p,A_2 \to B_2,? \to ?}[E_{e,A_1 \to B_2,? \to ?}] \end{aligned}$$

4. (Products) Same argument as function case.

5. $(A_1, A_2 \neq ? \wedge \lfloor A_1 \rfloor \neq \lfloor A_2 \rfloor) \quad E_{\langle A_2 \Leftarrow A_1 \rangle} \stackrel{\text{def}}{=} \text{let } x = [\cdot] \text{ in } \upsilon$ We choose $A' = ?$, so we need to show:

   $$\text{let } x = [\cdot] \text{ in } \upsilon \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{e,A_1,?}]$$

   By embedding, projection decomposition this is equivalent to

   $$\text{let } x = [\cdot] \text{ in } \upsilon \sqsupseteq\sqsubseteq E_{p,A_2,\lfloor A_2 \rfloor}[E_{p,\lfloor A_2 \rfloor,?}[E_{e,A_1,\lfloor A_1 \rfloor}[E_{e,A_1,\lfloor A_1 \rfloor}]]]$$

   Which holds by open β because the embedding $E_{e,A_1,\lfloor A_1 \rfloor}$ is pure and $\lfloor A_1 \rfloor \neq \lfloor A_2 \rfloor$.

□

And since the only difference between direct and ep pair semantics is the interpretation of casts, we get the equivalence between direct and ep pair semantics overall:

Corollary 56. *For any cast calculus term $\Gamma \vdash t : A$, $|\Gamma| \vDash [\![t]\!]^{dir} \sqsupseteq\sqsubseteq [\![t]\!]^{ep} : |A|$*

Proof. By induction over t, using either congruence or equivalence of the two cast semantics. □

3.5 SOUNDNESS OF $\beta\eta$ EQUALITY

Now that we have proved some lemmas about casts, we can easily prove the soundness of $\beta\eta$.

Theorem 57. *If $\Gamma \vdash t \equiv u : A$ in the cast calculus, then $|\Gamma| \vdash [\![t]\!]^{ep} \equiv [\![u]\!]^{ep} : |A|$*

Proof. Congruence cases follow from the congruence Theorem 27, β cases follow from Lemma 33, η cases follow from Lemma 34, reflexivity (Lemma 30) and transitivity (Lemma 32). Finally, the identity cast axiom follows from Lemma 47. □

Theorem 58. *If $\cdot \vdash t \equiv u : A$ in the cast calculus, then $t \simeq u$.*

Proof. First, by Theorem 57, $[\![t]\!]^{ep} \simeq [\![u]\!]^{ep}$. By Corollary 56, this means $[\![t]\!]^{dir} \simeq [\![u]\!]^{dir}$. Then by adequacy Theorem 23, this means $t \simeq u$. □

3.6 GRADUALITY FROM EP PAIRS

To prove the graduality theorem, we first prove that term precision in the cast calculus implies logical ordering of their translations. However, our logical relation is a *homogeneous* relation in that it only relates terms *of the same type*, whereas term precision relates terms where one has a more precise type (and context) than the other. So instead we prove (at the end of this section) that terms are ordered *up to* insertion of casts on the more precisely typed side. We define this as our notion of "logical term precision" for metalanguage terms:

Definition 59. Let $\Phi : \Gamma_1 \sqsubseteq \Gamma_2$ and $A_1 \sqsubseteq B$ hold in the cast calculus and let $|\Gamma_1| \vdash t_1 : |A_1|$ and $|\Gamma_2| \vdash t_2 : |A_2|$ be terms of the metalanguage. We define the logical term precision ordering as follows:

$$\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 = |\Gamma_2| \vDash \texttt{let}\ E_{p,[\![\Gamma_1]\!],[\![\Gamma_2]\!]} = |\Gamma_1|; E_{e,A_1,A_2}[t_1] \sqsubseteq t_2 : |A_2|$$

It is most natural to define this ordering by casting the more precisely typed term to the lest precise typing because we are interested in what happens when we migrate from a less precise typing to a more precise typing, in which case the more precise term will be placed in a context that expects the less precise typing.

Then our fundamental property for logical graduality is the following, which says that syntactic term precision in the cast calculus implies logical term precision of the translations.

Theorem 60 (Logical Graduality)**.** *If $\Phi \vdash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$, then*

$$\Phi \vDash [\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2$$

By adequacy, and the purity of embeddings, this is sufficient to prove graduality of the cast calculus.

Theorem 61 (Graduality). *If* $\cdot \vdash t \sqsubseteq u : A \sqsubseteq B$, *then* $t \sqsubseteq_{err} u$.

Proof. First, by logical graduality, we know

$$E_{e,A,B}[[\![t]\!]^{ep}] \sqsubseteq_{\text{err}} [\![u]\!]^{ep}$$

Since embeddings are pure, this means

$$[\![t]\!]^{ep} \sqsubseteq_{\text{err}} [\![u]\!]^{ep}$$

Then by adequacy of the ep pair semantics, we have

$$t \sqsubseteq_{\text{err}} u$$

. □

The remainder of this section is then dedicated to proving our logical graduality lemma. Now that we are in the realm of logical approximation, we have all the lemmas of §3.3.2 at our disposal, and we now start putting them to work. First, we show that at least with logical term precision, the decision to cast the more precise term to the type of the less precise term was arbitrary. Instead, we might have embedded the inputs on the right hand side or projected the output of the right, and all these definitions would be equivalent. The property we need is that the upcast and downcast are *adjoint* (in the language of category theory), also known as a *Galois connection*, which is a basic consequence of the definition of ep pair:

Lemma 62 (EP Pairs are Adjoint). *For any ep pair* $(E_e, E_p) : A_1 \lhd A_2$, *and terms* $\Gamma \vdash t_1 : A_1, , \Gamma \vdash t_2 : A_2$,

$$\Gamma \vDash E_e[t_1] \sqsubseteq^{log} t_2 : A_2 \quad \text{iff} \quad \Gamma \vDash t_1 \sqsubseteq^{log} E_p[t_2] : A_1$$

Proof. The two proofs are dual .

$$\dfrac{\dfrac{}{\Gamma \vDash t_1 \sqsubseteq^{\log} E_p[E_e[t_1]] : A_1}\ \textsc{EP Pair}}{}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vDash E_e[t_1] \sqsubseteq^{\log} t_2 : A_2}\ \textsc{Assumption}}{\Gamma \vDash E_p[E_e[t_1]] \sqsubseteq^{\log} E_p[t_2] : A_1}\ \textsc{Congruence}}{\Gamma \vDash t_1 \sqsubseteq^{\log} E_p[t_2] : A_1}\ \textsc{Transitivity}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vDash E_e[t_1] \sqsubseteq^{\log} E_p[t_2] : A_1}\ \textsc{Assumption}}{\Gamma \vDash t_1 \sqsubseteq^{\log} E_e[E_p[t_2]] : A_1}\ \textsc{Congruence}}{}$$

$$\dfrac{\dfrac{}{\Gamma \vDash E_e[E_p[t_2]] \sqsubseteq^{\log} t_2}\ \textsc{EP Pair}}{\Gamma \vDash E_e[t_1] \sqsubseteq^{\log} t_2 : A_2}\ \textsc{Transitivity}$$

□

$$\frac{\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq B_2 \qquad (A_2 \sqsubseteq B_2 \vee B_2 \sqsubseteq A_2)}{\Phi \vDash t_1 \sqsubseteq E_{\langle B_2 \Leftarrow A_2 \rangle}[t_2] : A_1 \sqsubseteq B_2} \textsc{Cast-Right}$$

$$\frac{\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq A_2 \qquad (A_1 \sqsubseteq B_1 \vee B_1 \sqsubseteq A_1)}{\Phi \vDash E_{\langle B_1 \Leftarrow A_1 \rangle}[t_1] \sqsubseteq t_2 : B_1 \sqsubseteq A_2} \textsc{Cast-Left}$$

Figure 3.15: Term Precision Upcast, Downcast Rules

Lemma 63 (Adjointness on Inputs). *If* $\Gamma, x_1 : A_1 \vdash t_1 : B$ *and* $\Gamma, x_2 : A_2 \vdash t_2 : B$, *and* $E_e, E_p : A_1 \lhd A_2$, *then*

$$\Gamma, x_1 : A_1 \vDash t_1 \sqsubseteq^{log} \text{let } x_2 = E_e[x_1] \text{ in } t_2 : B$$

iff

$$\Gamma, x_2 : A_2 \vDash \text{let } x_1 = E_p[x_2] \text{ in } t_1 \sqsubseteq^{log} t_2 : B$$

Proof. By a similar argument to lemma 62 □

This adjointness implies that our definition of logical term precision by embedding the more precise term with the more imprecise typing is equivalent to the 3 other apparent choices of definition based on whether we use the more or less precise input context and the more or less precise output type. This allows freedom in proofs to use whichever is most convenient for the case at hand.

Lemma 64 (Alternative Formulations of Logical Term Precision). *The following are all equivalent to* $\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2$ *(where* $\Phi : \Gamma_1 \sqsubseteq \Gamma_2$*)*

1. $|\Gamma_1| \vDash E_{e,A_1,A_2} t_1 \sqsubseteq^{log} \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } t_2 : |A_2|$
2. $|\Gamma_1| \vDash t_1 \sqsubseteq^{log} \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } E_{p,A_1,A_2} t_2 : |A_2|$
3. $|\Gamma_1| \vDash \text{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \text{ in } t_1 \sqsubseteq^{log} E_{p,A_1,A_2} t_2 : |A_2|$
4. $|\Gamma_1| \vDash E_{e,A_1,A_2} \text{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \text{ in } t_1 \sqsubseteq^{log} t_2 : |A_2|$

Proof. By induction on Γ_1, using lemma 62 and Lemma 63 □

Finally, to prove the graduality theorem, we do an induction over all the cases of syntactic term precision. Most important is the cast case $\langle B_1 \Leftarrow A_1 \rangle t_1 \sqsubseteq \langle B_2 \Leftarrow A_2 \rangle t_2$ which is valid when $A_1 \sqsubseteq A_2$ and $B_1 \sqsubseteq B_2$. We break up the proof into 4 atomic steps using the factorization of general casts into an upcast followed by a downcast (lemma 55): $E_{\langle A_2 \Leftarrow A_1 \rangle} \sqsupseteq\sqsubseteq E_{p,A_2,?}[E_{e,A_1,?}]$. The four steps are upcast on the left, downcast on the left, upcast on the right, and downcast on the right. These are presented as rules for logical term precision in Figure 3.15. Each of the inference rules accounts for two cases. The Cast-Right

rule says first that if $t_1 \sqsubseteq^{\log} t_2 : A_1 \sqsubseteq A_2$ that it is OK to cast t_2 to B_2, as long as B_2 is more dynamic than A_1, and the cast is either an upcast or downcast. Here, our explicit inclusion of $A_1 \sqsubseteq B_2$ in the syntax of the term precision judgment should help: the rule says that adding an upcast or downcast to t_2 results in a more dynamic term than t_1, *whenever it is even sensible to ask*: i.e., if it were not the case that $A_1 \sqsubseteq B_2$, the judgment would not be well-formed, so the judgment holds whenever it makes sense! The Cast-Left rule is dual.

These 4 rules, combined with our factorization of casts into upcast followed by downcast suffice to prove the congruence rule for casts:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2}{[\![t_1]\!] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : A_1 \sqsubseteq ?}\ \textsc{Cast-Right}}{E_{e,A_1,?}[[\![t_1]\!]] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : ? \sqsubseteq ?}\ \textsc{Cast-Left}}{E_{p,B_1,?}[E_{e,A_1,?}[[\![t_1]\!]]] \sqsubseteq E_{e,A_2,?}[[\![t_2]\!]] : B_1 \sqsubseteq ?}\ \textsc{Cast-Left}}{E_{p,B_1,?}[E_{e,A_1,?}[[\![t_1]\!]]] \sqsubseteq E_{p,B_2,?}[E_{e,A_2,?}[[\![t_2]\!]]] : B_1 \sqsubseteq B_2}\ \textsc{Cast-Right}}{\Phi \vDash [\![\langle B_1 \Leftarrow A_1 \rangle t_1]\!] \sqsubseteq [\![\langle B_2 \Leftarrow A_2 \rangle t_2]\!] : B_1 \sqsubseteq B_2}\ \textsc{lemma}\ 55$$

Next, we show the 4 rules are valid, as simple consequences of the ep pair property and the decomposition theorem. Also note that while there are technically 4 cases, each comes in a pair where the proofs are exactly dual, so conceptually speaking there are only 2 arguments.

Lemma 65 (Upcast, Downcast Precision)**.** *The four rules in Figure 3.15 are valid.*

Proof. In each case we choose which case of lemma 64 is simplest.

1. Cast-Left with $A_1 \sqsubseteq B_1 \sqsubseteq A_2$. We need to show $E_{e,B_1,A_2}[E_{e,A_1,B_1}[t_1]] \sqsubseteq$ let $[\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]$ in t_2. By decomposition and congruence,

   $$E_{e,B_1,A_2}[E_{e,A_1,B_1}[t_1]] \sqsupseteq\sqsubseteq E_{e,A_1,A_2}$$

   so the conclusion holds by transitivity and the premise.

2. Cast-Right with $A_1 \sqsubseteq B_2 \sqsubseteq A_2$. We need to show let $[\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]]$ in $t_1 \sqsubseteq E_{p,A_1,B_2}[E_{p,B_2,A_2}[t_2]]$. By decomposition and congruence,

   $$E_{p,A_1,B_2}[E_{p,B_2,A_2}[t_2]] \sqsupseteq\sqsubseteq E_{p,A_1,A_2}[t_2]$$

   , so the conclusion holds by transitivity and the premise.

3. Cast-Left with $B_1 \sqsubseteq A_1 \sqsubseteq A_2$. We need to show

   $$E_{p,B_1,A_1}[\text{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \text{ in } t_1] \sqsubseteq E_{p,B_1,A_2}[t_2]$$

. By decomposition, $E_{p,B_1,A_2}[t_2] \sqsupseteq\sqsubseteq E_{p,B_1,A_1}[E_{p,A_1,A_2}[t_2]]$, so by transitivity it is sufficient to show

$$E_{p,B_1,A_1}[\text{let } [\![\Gamma_1]\!] = E_{p,\Gamma_1,\Gamma_2}[[\![\Gamma_2]\!]] \text{ in } t_1] \sqsubseteq E_{p,B_1,A_1}[E_{p,A_1,A_2}[t_2]]$$

which follows by congruence and the premise.

4. Cast-Right with $A_1 \sqsubseteq A_2 \sqsubseteq B_2$. We need to show $E_{e,A_1,B_2}[t_1] \sqsubseteq E_{e,A_2,B_2}[\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } t_2]$.

   By decomposition, $E_{e,A_1,B_2}[t_1] \sqsupseteq\sqsubseteq E_{e,A_2,B_2}[E_{e,A_1,A_2}[t_1]]$, so by transitivity it is sufficient to show

$$E_{e,A_2,B_2}[E_{e,A_1,A_2}[t_1]] \sqsubseteq E_{e,A_2,B_2}[\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } t_2]$$

   which follows by congruence and the premise.

□

Finally, we prove the graduality theorem by induction on syntactic term precision derivations, finishing the proof of logical graduality.

Proof of Theorem 60. By induction on syntactic term precision rules, we show that if $\Gamma_1 \sqsubseteq \Gamma_2 \vdash t_1 \sqsubseteq t_2 : A_1$, then $\Phi \vDash [\![t_1]\!] \sqsubseteq [\![t_2]\!] : A_1 \sqsubseteq A_2$.

1. To show $\dfrac{\Phi \vDash [\![t_1]\!] \sqsubseteq [\![t_2]\!] : B_1 \sqsubseteq B_2}{\Phi \vDash [\![\langle B_1 \Leftarrow A_1\rangle t_1]\!] \sqsubseteq [\![\langle B_2 \Leftarrow A_2\rangle t_2]\!] : B_1 \sqsubseteq B_2}$ we use lemma 65 and the argument above.

2. $\dfrac{\Phi \ni x_1 \sqsubseteq x_2 : A_1 \sqsubseteq A_2}{\Phi \vDash x_1 \sqsubseteq x_2 : A_1 \sqsubseteq A_2}$ We need to show:

$$[\![\Gamma_1]\!] \vDash E_{e,A_1,A_2}[x_1] \sqsubseteq \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } x_2$$

   Since embeddings are pure (Lemmas lemmas 40 and 43) we can substitute them in and then the two sides are literally the same.

$$\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } x_2 \sqsupseteq\sqsubseteq x_2[E_{e,A_1^i,A_2^i}[x_1^i]/x_2^i] = E_{e,A_1,A_2}[x_1]$$

3. $\Phi \vDash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool} \sqsubseteq \texttt{Bool}$. Expanding definitions, we need to show that

$$E_{e,\texttt{Bool},\texttt{Bool}}[\texttt{true}] \sqsubseteq \texttt{true}$$

   which follows immediately from the definition because $E_{e,\texttt{Bool},\texttt{Bool}} = [\cdot]$

4. $\Phi \vDash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool} \sqsubseteq \texttt{Bool}$ Analogous to previous case.

5. $$\frac{\Phi \vDash t_1 \sqsubseteq t_2 : \mathtt{Bool} \sqsubseteq \mathtt{Bool} \qquad \Phi \vDash s_1 \sqsubseteq s_2 : B_1 \sqsubseteq B_2 \qquad \Phi \vDash s_1' \sqsubseteq s_2' : B_1 \sqsubseteq B_2}{\Phi \vDash \mathtt{if}\ t_1\ \mathtt{then}\ s_1\ \mathtt{else}\ s_1' \sqsubseteq \mathtt{if}\ t_2\ \mathtt{then}\ s_2\ \mathtt{else}\ s_2' : B_1 \sqsubseteq B_2}$$ Expanding definitions, we need to show

$$E_{e,B_1,B_2}[\mathtt{if}\ [\![t_1]\!]\ \mathtt{then}\ [\![s_1]\!]\ \mathtt{else}\ [\![s_1']\!]] \sqsubseteq \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in}\ \mathtt{if}\ [\![t_2]\!]\ \mathtt{then}\ [\![s_2]\!]\ \mathtt{else}\ [\![s_2']\!]$$

First, we do some simple rewrites: on the left side, we use a commuting conversion to push the embedding into the continuations:

$$E_{e,B_1,B_2}[\mathtt{if}\ [\![t_1]\!]\ \mathtt{then}\ [\![s_1]\!]\ \mathtt{else}\ [\![s_1']\!]] \sqsupseteq\sqsubseteq \mathtt{if}\ [\![t_1]\!]\ \mathtt{then}\ E_{e,B_1,B_2}[\![s_1]\!]\ \mathtt{else}\ E_{e,B_1,B_2}[\![s_1']\!]$$

And on the right side we use the fact that embeddings are pure and so can be moved freely:

$$\begin{array}{c} \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in} \\ \mathtt{if}\ [\![t_2]\!]\ \mathtt{then}\ [\![s_2]\!]\ \mathtt{else}\ [\![s_2']\!] \end{array}$$

$$\sqsupseteq\sqsubseteq$$

$$\mathtt{if}\ \begin{pmatrix} \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in} \\ [\![t_2]\!] \end{pmatrix}\ \mathtt{then}\ \begin{pmatrix} \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in} \\ [\![s_2]\!] \end{pmatrix}\ \mathtt{else}\ \begin{pmatrix} \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in} \\ [\![s_2]\!] \end{pmatrix}$$

So the result follows by congruence and inductive hypothesis.

6. $$\frac{\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \sqsubseteq A_2 \qquad \Phi \vDash s_1 \sqsubseteq s_2 : B_1 \sqsubseteq B_2}{\Phi \vDash \langle t_1, s_1 \rangle \sqsubseteq \langle t_2, s_2 \rangle : A_1 \times B_1 \sqsubseteq A_2 \times B_2}$$. Expanding definitions, we need to show

$$E_{e,A_1\times B_1,A_2\times B_2}\langle [\![t_1]\!], [\![s_1]\!] \rangle \sqsubseteq \mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in}\ \langle [\![t_2]\!], [\![s_2]\!] \rangle$$

On the right, we duplicate the embeddings, justified by lemmas 40 and 43, to set up congruence:

$$\mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in} \sqsupseteq\sqsubseteq \langle \underbrace{\mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in}}_{[\![t_2]\!]}, \underbrace{\mathrm{let}\ [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\ \mathrm{in}}_{[\![s_2]\!]} \rangle$$
$$\langle [\![t_2]\!], [\![s_2]\!] \rangle$$

On the left, we use linearity of evaluation contexts to lift the terms out, then perform some open β reductions and put the terms back in:

$$\begin{aligned}
E_{e,A_1\times B_1,A_2\times B_2}\langle [\![t_1]\!], [\![s_1]\!]\rangle &\sqsupseteq\sqsubseteq \text{let } x = [\![t_1]\!] \text{ in} \\
&\quad \text{let } y = [\![s_1]\!] \text{ in} \\
&\quad \text{let } \langle x, y\rangle = \langle x, y\rangle \text{ in } \langle E_{e,A_1,A_2}x, E_{e,B_1,B_2}y\rangle \\
(\text{open } \beta,\ 33) &\sqsupseteq\sqsubseteq \text{let } x = [\![t_1]\!] \text{ in} \\
&\quad \text{let } y = [\![s_1]\!] \text{ in} \\
&\quad \langle E_{e,A_1,A_2}x, E_{e,B_1,B_2}y\rangle \\
(\text{linearity},\ 36) &\sqsupseteq\sqsubseteq \langle E_{e,A_1,A_2}[\![t_1]\!], E_{e,B_1,B_2}[\![s_1]\!]\rangle
\end{aligned}$$

With the final step following by congruence (lemma 27) and the premise:

$$\langle E_{e,A_1,A_2}[\![t_1]\!], E_{e,B_1,B_2}[\![s_1]\!]\rangle \sqsubseteq \langle \underset{[\![t_2]\!]}{\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in}}, \underset{[\![s_2]\!]}{\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in}}\rangle$$

7. $$\dfrac{\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \times A_1' \sqsubseteq A_2 \times A_2' \qquad \Phi, x_1 \sqsubseteq x_2 : A_1 \sqsubseteq A_2 \vDash t_1' \sqsubseteq t_2' : B_1 \sqsubseteq B_2}{\Phi \vDash \begin{array}{c}\text{let } \langle x_1 : A_1, x_1' : A'\rangle = t_1 \text{ in } t_1' \\ \sqsubseteq \\ \text{let } \langle x_2 : A_2, x_2' : A_2'\rangle = t_2 \text{ in } t_2'\end{array} : B_1 \sqsubseteq B_2}$$ Expanding definitions, we need to show

$$[\![\Gamma_1]\!] \vDash E_{e,B_1,B_2}\text{let } \langle x_1, x_1'\rangle = [\![t_1]\!] \text{ in } [\![t_1']\!] \sqsubseteq \underset{\text{let } \langle x_2, x_2'\rangle = [\![t_2]\!] \text{ in } [\![t_2']\!]}{\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } : [\![B_2]\!]}$$

On the right side, in anticipation of a use of congruence, we push the embeddings in lemmas 40 and 43:

$$\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in } \sqsupseteq\sqsubseteq \text{let } \langle x_2, x_2'\rangle = \left(\begin{array}{l}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![t_2]\!]\end{array}\right) \text{ in}$$
$$\text{let } \langle x_2, x_2'\rangle = [\![t_2]\!] \text{ in } [\![t_2']\!]$$
$$\begin{array}{l}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![t_2']\!]\end{array}$$

On the left side, we perform a commuting conversion, ep expand the discriminee and do some open β reductions to simplify the expression.

$$\begin{aligned}
& E_{e,B_1,B_2}\mathsf{let}\,\langle x_1, x_1'\rangle = [\![t_1]\!]\,\mathsf{in}\,[\![t_1']\!] \\
\sqsupseteq\sqsubseteq\ & \mathsf{let}\,\langle x_1, x_1'\rangle = [\![t_1]\!]\,\mathsf{in}\,E_{e,B_1,B_2}[\![t_1']\!] \\
(\text{ep pair, } 14)\ \sqsupseteq\sqsubseteq\ & \mathsf{let}\,\langle x_1, x_1'\rangle = E_{p,A_1\times A_1',A_2\times A_2'}E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in} \\
& \quad E_{e,B_1,B_2}[\![t_1']\!] \\
(\text{definition})\ =\ & \mathsf{let}\,\langle x_1, x_1'\rangle = \mathsf{let}\,\langle x_2, x_2'\rangle = E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in}\,\mathsf{in} \\
& \quad \langle E_{p,A_1,A_2}x_2, E_{p,A_1',A_2'}x_2\rangle \\
& E_{e,B_1,B_2}[\![t_1']\!] \\
(\text{linearity, } 36)\ \sqsupseteq\sqsubseteq\ & \mathsf{let}\,\langle x_1, x_1'\rangle = \mathsf{let}\,\langle x_2, x_2'\rangle = E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in}\,\mathsf{in} \\
& \quad \mathsf{let}\,x_1 = E_{p,A_1,A_2}x_2\,\mathsf{in} \\
& \quad \mathsf{let}\,x_1' = E_{p,A_1',A_2'}x_2'\,\mathsf{in} \\
& \quad \langle x_1, x_1'\rangle \\
& E_{e,B_1,B_2}[\![t_1']\!] \\
(\text{comm. conv., } 35)\ \sqsupseteq\sqsubseteq\ & \mathsf{let}\,\langle x_2, x_2'\rangle = E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in} \\
& \mathsf{let}\,x_1 = E_{p,A_1,A_2}x_2\,\mathsf{in} \\
& \mathsf{let}\,x_1' = E_{p,A_1',A_2'}x_2'\,\mathsf{in} \\
& \mathsf{let}\,\langle x_1, x_1'\rangle = \langle x_1, x_1'\rangle\,\mathsf{in} \\
& E_{e,B_1,B_2}[\![t_1']\!] \\
(\text{open } \beta, \text{ } 33)\ \sqsupseteq\sqsubseteq\ & \mathsf{let}\,\langle x_2, x_2'\rangle = E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in} \\
& \mathsf{let}\,x_1 = E_{p,A_1,A_2}x_2\,\mathsf{in} \\
& \mathsf{let}\,x_1' = E_{p,A_1',A_2'}x_2'\,\mathsf{in} \\
& E_{e,B_1,B_2}[\![t_1']\!]
\end{aligned}$$

The final step is by congruence and adjointness on inputs (lemmas 27 and 63):

$$\begin{array}{l}
\mathsf{let}\,\langle x_2, x_2'\rangle = E_{e,A_1\times A_1',A_2\times A_2'}[\![t_1]\!]\,\mathsf{in} \\
\mathsf{let}\,x_1 = E_{p,A_1,A_2}x_2\,\mathsf{in} \\
\mathsf{let}\,x_1' = E_{p,A_1',A_2'}x_2'\,\mathsf{in} \\
E_{e,B_1,B_2}[\![t_1']\!]
\end{array}
\sqsubseteq
\mathsf{let}\,\langle x_2, x_2'\rangle = \left(\begin{array}{l}\mathsf{let}\,[\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\,\mathsf{in} \\ [\![t_2]\!]\end{array}\right)\,\mathsf{in}
\begin{array}{l}
\\
\\
\\
\mathsf{let}\,[\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\,\mathsf{in} \\
[\![t_2']\!]
\end{array}$$

8. $\dfrac{\Phi, x_1 \sqsubseteq x_2 : A_1 \sqsubseteq A_1 \vDash t_1 \sqsubseteq t_2 : B_1 \sqsubseteq B_2}{\Phi \vDash \lambda(x_1 : A_1).\,t_1 \sqsubseteq \lambda(x_2 : A_2).\,t_2 : A_1 \to B_1 \sqsubseteq A_2 \to B_2}$. Expanding definitions, we need to show

$$\begin{array}{l}
[\![\Gamma_1]\!] \vDash \mathsf{let}\,x_f = \lambda(x_1 : [\![A_1]\!]).\,[\![t_1]\!]\,\mathsf{in} \\
\qquad \lambda(x_2 : A_2).\,E_{e,B_1,B_2}x_f\,(E_{p,A_1,A_2}x_2)
\end{array}
\sqsubseteq
\begin{array}{l}
\mathsf{let}\,[\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]]\,\mathsf{in} \\
\lambda(x_2 : A_2).\,[\![t_2]\!]
\end{array}$$

First we simplify by performing some open β reductions on the left and let-λ equivalence and a commuting conversion (lemmas 33, 35 and 37):

$$\begin{array}{l}\text{let } x_f = \lambda(x_1 \,:\, [\![A_1]\!]).\, [\![t_1]\!] \text{ in} \\ \lambda(x_2 \,:\, A_2).\, E_{e,B_1,B_2}\, x_f\, (E_{p,A_1,A_2}\, x_2) \\ \sqsupseteq\sqsubseteq \lambda(x_2 \,:\, A_2).\, E_{e,B_1,B_2} \text{let } x_1 = E_{p,A_1,A_2}\, x_2 \text{ in} \\ \qquad\qquad\qquad\qquad [\![t_1]\!] \\ \sqsupseteq\sqsubseteq \lambda(x_2 \,:\, A_2).\, \text{let } x_1 = E_{p,A_1,A_2}\, x_2 \text{ in} \\ \qquad\qquad\qquad\qquad E_{e,B_1,B_2} [\![t_1]\!]\end{array}$$

and on the right, we move the embedding into the body, which is justified because embeddings are essentially values (lemmas 40 and 43):

$$\begin{array}{ll}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} & \sqsupseteq\sqsubseteq \lambda(x_2 \,:\, A_2).\, \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ \lambda(x_2 \,:\, A_2).\, [\![t_2]\!] & \qquad\qquad\qquad\qquad\qquad\qquad [\![t_2]\!]\end{array}$$

The final step is justified by congruence lemma 27 and adjointness on inputs lemma 63 and the premise:

$$\begin{array}{ll}\lambda(x_2 \,:\, A_2).\, \text{let } x_1 = E_{p,A_1,A_2}\, x_2 \text{ in} & \sqsupseteq\sqsubseteq \lambda(x_2 \,:\, A_2).\, \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ \qquad E_{e,B_1,B_2} [\![t_1]\!] & \qquad\qquad\qquad\qquad\qquad\qquad [\![t_2]\!]\end{array}$$

9. $\dfrac{\Phi \vDash t_1 \sqsubseteq t_2 : A_1 \to B_1 \sqsubseteq A_2 \to B_2 \qquad \Phi \vDash s_1 \sqsubseteq s_2 : A_1 \sqsubseteq A_2}{\Phi \vDash t_1\ s_1 \sqsubseteq t_2\ s_2 : B_1 \sqsubseteq B_2}$. Expanding definitions, we need to show

$$\begin{array}{c}[\![\Gamma_1]\!] \vDash E_{e,B_1,B_2} [\![t_1]\!]\, [\![s_1]\!] \sqsubseteq \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} : [\![B_2]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad [\![t_2]\!]\, [\![s_2]\!]\end{array}$$

First, we duplicate the embedding on the right hand side, justified by purity of embeddings, to set up a use of congruence later:

$$\begin{array}{c}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![t_2]\!]\, [\![s_2]\!]\end{array}$$

$$\sqsupseteq\sqsubseteq$$

$$\left(\begin{array}{l}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![t_2]\!]\end{array}\right) \left(\begin{array}{l}\text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![s_2]\!]\end{array}\right)$$

Next, we use linearity of evaluation contexts lemma 36 so that we can do reductions at the application site without worrying about evaluation order:

$$\begin{aligned} E_{e,B_1,B_2}[\![t_1]\!]\,[\![s_1]\!] \sqsupseteq\sqsubseteq\ & \text{let } x_f = [\![t_1]\!] \text{ in} \\ & \text{let } x_a = [\![s_1]\!] \text{ in} \\ & E_{e,B_1,B_2}\, x_f\, x_a \end{aligned}$$

Next, we ep-expand x_f (theorem 14) and perform some β reductions, use the ep property and then reverse the use of linearity.

$$\begin{aligned} \text{let } x_f = [\![t_1]\!] \text{ in} & \sqsupseteq\sqsubseteq \text{let } x_f = [\![t_1]\!] \text{ in} \\ \text{let } x_a = [\![s_1]\!] \text{ in} & \qquad \text{let } x_a = [\![s_1]\!] \text{ in} \\ E_{e,B_1,B_2}\, x_f\, x_a & \qquad E_{e,B_1,B_2} E_{p,A_1\to B_1,A_2\to B_2} E_{e,A_1\to B_1,A_2\to B_2}\, x_f\, x_a \\ (\text{open } \beta,\ 33) & \sqsupseteq\sqsubseteq \text{let } x_f = [\![t_1]\!] \text{ in} \\ & \qquad \text{let } x_a = [\![s_1]\!] \text{ in} \\ & \qquad E_{e,B_1,B_2} E_{p,B_1,B_2} \left(E_{e,A_1\to B_1,A_2\to B_2}\, x_f\right)\ \left(E_{e,A_1,A_2}\, x_a\right) \\ (\text{ep pair},\ 14) & \sqsubseteq \text{let } x_f = [\![t_1]\!] \text{ in} \\ & \qquad \text{let } x_a = [\![s_1]\!] \text{ in} \\ & \qquad \left(E_{e,A_1\to B_1,A_2\to B_2}\, x_f\right)\ \left(E_{e,A_1,A_2}\, x_a\right) \\ (\text{linearity},\ 36) & \sqsupseteq\sqsubseteq \left(E_{e,A_1\to B_1,A_2\to B_2}[\![t_1]\!]\right)\ \left(E_{e,A_1,A_2}[\![s_1]\!]\right) \end{aligned}$$

With the final step being congruence lemma 27:

$$\left(E_{e,A_1\to B_1,A_2\to B_2}[\![t_1]\!]\right)\ \left(E_{e,A_1,A_2}[\![s_1]\!]\right)$$

$$\sqsupseteq\sqsubseteq \begin{pmatrix} \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![t_2]\!] \end{pmatrix} \begin{pmatrix} \text{let } [\![\Gamma_2]\!] = E_{e,\Gamma_1,\Gamma_2}[[\![\Gamma_1]\!]] \text{ in} \\ [\![s_2]\!] \end{pmatrix}$$

□

3.7 RELATED WORK AND DISCUSSION

DIFFERENCES WITH NEW AND AHMED [56] The presentation here is mostly the same as in the original paper New and Ahmed [56], but there are a few notable differences. First, I model surface syntax and reasoning about surface programs in the previous chapter, since this is where programmer reasoning is relevant, including explicitly describing $\beta\eta$ equality. Second, the surface calculi here have only booleans, since these are typically supported by a dynamic language, whereas in that paper we supported arbitrary sums. Finally, the original paper heavily used intermediate semantic notions of contextual equivalence and contextual error approximation, and defines the graduality theorem in terms of these. Instead I simply define semantic

equivalence and approximation for closed terms, and then define a logical relation that is sound with respect to this notion. Since our syntactic relations of $\beta\eta$ equivalence and term precision are closed under program contexts, defining this intermediate notion is unnecessary.

TYPES AS EP PAIRS The interpretation of types as retracts of a single domain originated in Scott [67] and is a common tool in denotational semantics, especially in the presence of a convenient *universal* domain. A retraction is a pair of morphisms $s : A \to B$, $r : B \to A$ that satisfy the retraction property $r \circ s = \mathrm{id}_A$, but not necessarily the projection property $s \circ r \sqsubseteq_{\mathrm{err}} \mathrm{id}_B$. Thus ep pair semantics can be seen as a more refined retraction semantics. Retractions have been used to study interaction between typed and untyped languages, e.g., see Benton [9] and (Favonia), Benton, and Harper [21].

Embedding-projection pairs are used extensively in domain theory as a technical device for solving non-well-founded domain equations, such as the semantics of a dynamic type. In this paper, our error-approximation ep pairs do not play this role, and instead the retraction and projection properties are desirable in their own right for their intuitive meaning for type checking.

Many of the properties of our embedding-projection pairs are anticipated in Henglein [38] and Thatte [80]. Henglein [38] defines a language with a notion of *coercion* $A \rightsquigarrow B$ that corresponds to general casts, with primitives of tagging $tc! : tc(?,\ldots) \rightsquigarrow ?$ and untagging $tc? : ? \rightsquigarrow tc(?,\ldots)$ for every type constructor "tc". Crucially, Henglein notes that $tc!; tc?$ is the identity modulo efficiency and that $tc?; tc!$ errors more than the identity. Furthermore, they define classes of "positive" and "negative" coercions that correspond to embeddings and projections, respectively, and a "subtyping" relation that is the same as type precision. They then prove several theorems analogous to our results:

1. (Retraction) For any pair of positive coercion $p : A \rightsquigarrow B$, and negative coercion $n : B \rightsquigarrow A$, they show that $p; n$ is equal to the identity in their equational theory.

2. (Almost projection) Dually, they show that $n; p$ is equal to the identity *assuming* that $tc?; tc!$ is equal to the identity for every type constructor.

3. They show every coercion factors as a positive cast to ? followed by a negative cast to ?.

4. They show that $A \leq B$ if and only if there exists a positive coercion $A \rightsquigarrow B$ and a negative coercion $B \rightsquigarrow A$.

They also prove factorization results that are similar to our factorization definition of semantic type precision, but it is unclear if their theorem is stronger or weaker than ours. One major difference is that

their work is based on an equational theory of casts, whereas ours is based on notions of observational equivalence and approximation of a standard call-by-value language. Furthermore, in defining our notion of observational error approximation, we provide a more refined projection property, justifying their use of the term "safer" to compare $p;e$ and the identity.

The system presented in Thatte [80], called "quasi-static typing" is a precursor to gradual typing that inserts type annotations into dynamically typed programs to make type checking explicit. There they prove a desirable soundness theorem that says their type insertion algorithm produces an explicitly coercing term that is minimal in that it errors no more than the original dynamic term. They prove this minimality theorem with respect to a partial order $\sqsupseteq$ defined as a logical relation over a domain-theoretic semantics that (for the types they defined) is analogous to our error ordering for the operational semantics. However, they do not define our operational formulation of the ordering as contextual approximation, linked to the denotational definition by the adequacy result, nor that any casts form embedding-projection pairs with respect to this ordering.

Finally, we note that neither of these papers [38, 80] extends the analysis to anything like graduality.

SEMANTICS OF CASTS Superficially similar to the embedding-projection pair semantics are the *threesome casts* of Siek and Wadler [72]. A threesome cast factorizes an arbitrary cast $A \Rightarrow B$ through a third type C as a *downcast* $A \Rightarrow C$ followed by an *upcast* $C \Rightarrow B$, whereas ep-pair semantics factorizes a cast as an *upcast* $A \Rightarrow ?$ followed by a *downcast* $? \Rightarrow B$. Threesome casts can be used to implement gradual typing in a space-efficient manner, the third type C is used to collapse a sequence of arbitrarily many casts into just the two. EP-pair semantics does not directly provide an efficient implementation, in fact it is one of the most naïve implementation techniques. However, we have shown that using ep pairs helps prove graduality, and we view them as the nice *specification*, whereas threesomes are a principled *implementation* technique.

Recently, work on *dependent interoperability* [15, 16] has identified Galois connections as a semantic formulation for casting between more and less precise types in a non-gradual dependently typed language, and conjectures that this should relate to type precision. We confirm their conjecture in showing that the casts in gradual typing satisfy the slightly stronger property of being embedding-projection pairs and used it to explain the cast semantics of gradual typing and graduality.

PAIRS OF PROJECTIONS AND BLAME One of the main inspirations for this work is the analysis of contracts in Findler and Blume [24]. Most relevant for our purposes is the notion of contracts as

error projections and the refinement to *pairs of projections*. While simple assertion-based contracts are presented by boolean predicates, higher-order contracts instead *coerce* values to behave according to the specification [24].

An error projection c on a type B is a function from B to itself satisfying two properties:

1. Idempotence: $c \circ c = c$
2. Projection: $c \sqsubseteq_{\text{err}} \texttt{id}$

The intuition is that c "forces" a value of B to behave according to some specification. Idempotence says that once something is forced to satisfy the specification, forcing it again makes no observable difference because it already satisfies the specification. Projection is analogous to graduality: it says that enforcing a contract only adds errors to a value's behavior. Here $\sqsubseteq_{\text{err}}$ is not formally defined but is analogous to the error ordering in the graduality theorem.

There is a close technical connection between error projections and embedding-projection pairs. First, from any embedding-projection pair from A to B I can construct an error projection on B. Given an embedding $e : A \to B$ and corresponding projection $p : B \to A$, the composite $e \circ p : B \to B$ is an error projection. Idempotence follows from the retraction property, and projection from the projection property of ep pairs. In a sufficiently rich semantic setting, any error projection c can be "split" into an embedding-projection pair from B_c to B, where the elements of the domain B_c are defined to be the elements of B that are invariant under the projection: $c(b) = b$. Intuitively these elements are the ones that "satisfy" the contract: invariance means that applying the contract results in no observable change in behavior. This constitutes an equivalence between the concepts of an ep pair and an error projection, so my work can be seen as a continuation of this work that emphasizes a more intrinsically typed view of contracts. Furthermore, Findler and Blume [24] models contracts as *pairs* of projections, one of which enforces the positive aspect of the contract and the other the negative. I argue that this idea is better modeled by embedding-projection pairs: the embedding enforces the negative aspects and the projection the positive aspects, but similar to "manifest" contracts [32] this is encoded in the type system so the embedding and projection are typed and mediate between a more precise and less precise type.

Findler and Blume's analysis of blame was adapted to gradual typing in Wadler and Findler [88] and plays a complementary role to our analysis: they use the precision relation to help prove the blame soundness theorem, whereas we use it to prove graduality. We discuss the relationship between blame safety and purity of embeddings/linearity of projections in Chapter 5

Part II

GRADUAL TYPE THEORY: AXIOMATIZING GRADUAL TYPING

4

INTRODUCTION TO PART II

We have now seen how embedding-projection pairs can provide a basis for the semantics of a simple gradually typed language that satisfies graduality and type-based reasoning principles. In the next chapter, we explore the extent to which these reasoning principles limit the design space of gradual typing: what design choices are left to make once we are committed to graduality and $\beta\eta$ equality? It turns out that the "wrapping" semantics of casts is the only semantics that satisfies graduality and $\beta\eta$ equality. This theorem can be employed to produce violations of $\beta\eta$ equality in alternative semantics that satisfy graduality.

We prove this theorem by taking an axiomatic approach to gradual typing, i.e. axiomatizing the semantic term precision property to additionally be closed under our desired $\beta\eta$ equalities. As mentioned previously, $\beta\eta$ equality is quite sensitive to evaluation order, so to maximize the generality of our result, we base our axiomatics on *Call-by-push-value* (CBPV), a metalanguage that includes as fragments call-by-value, call-by-name and "lazy" evaluation orders.

In Chapter 5, we review our framework, called *Gradual Type Theory* (GTT) and show how the semantics of casts are derivable from $\beta\eta$ equality. Then in Chapter 6, we show how we can use the standard elaborations of call-by-value, call-by-name and "lazy" evaluation orders into GTT to show how the single theory of GTT reproduces previously disparate approaches to cast semantics for different evaluation orders. Finally, in Chapter 7, we construct operational models of GTT by translation to CBPV.

Chapters 5 and 7 are based on New, Licata, and Ahmed [60], while Chapter 6 is novel material.

5

GRADUAL TYPE THEORY

5.1 GOALS

Note: the contents of this section are based on the paper Gradual Type Theory *co-authored with Amal Ahmed and Daniel Licata and published at POPL 2018*

In the previous chapters I have shown that the view of gradual types as embedding-projection pairs into the dynamic type helps to prove type soundness and graduality of a gradual language, i.e., that the theory of embedding-projection pairs is sufficient to develop the metatheory of some sound gradual languages. The goal of the work presented in this section is to study the converse: to what extent is the theory of embedding-projection pairs *inherent to* the study of sound gradually typed languages? We will also address another fundamental question: how much is the design of gradually typed languages constrained by the requirements of $\beta\eta$ equality and graduality? The gradual typing and contract literature exhibits a great deal of work on the "design space" of cast semantics. Which of these designs validate our desired reasoning principles?

These seemingly disparate questions will both be addressed using the same technique: an axiomatization of the reasoning principles we desire: graduality and $\beta\eta$ equivalence. Then we will see the embedding-projection pair semantics is a natural consequence of graduality, and that in some cases, the behavior of casts is *uniquely determined* by the constraints of $\beta\eta$ equality and graduality.

5.1.1 *Exploring the Design Space*

The linchpin to the design of a gradually typed language is the semantics of the casts. These runtime checks ensure that typed reasoning principles are valid by checking types of dynamically typed code at the boundary between static and dynamic typing. For instance, when a statically typed function $f : \mathtt{Num} \to \mathtt{Num}$ is applied to a dynamically typed argument $x : ?$, the language runtime must check if x is a number, and otherwise raise a dynamic type error. A programmer familiar with dynamically typed programming might object that this is overly strong: for instance, if f is just a constant function $f = \lambda x : \mathtt{Num}.0$ then why bother checking if x is a number since the body of the program does not seem to depend on it? The reason the value is rejected is because the annotation $x : \mathtt{Num}$ should introduce an assumption that that the programmer, compiler and automated tools can rely on for behavioral reasoning in the body of the function. For instance, if the variable x is guaranteed to only be instantiated with numbers, then the programmer is free to replace 0 with $x - x$ or vice-versa. However, if x

can be instantiated with a closure, then $x - x$ will raise a runtime type error while 0 will succeed, violating the programmers intuition about the correctness of refactorings. We can formalize such relationships by *observational equivalence* of programs: the two closures $\lambda x : \mathtt{Num}.0$ and $\lambda x : \mathtt{Num}.x - x$ are indistinguishable to any other program in the language.

The above is precisely the difference between gradual typing and so-called *optional* typing: in an optionally typed language (Hack, TypeScript, Flow), annotations are checked for consistency but are unreliable to the user, so do not provide a sound foundation for reasoning in general. We have very much of the syntactic discipline of static typing but very little of the semantic rewards. In a gradually typed language, type annotations should relieve the programmer of the burden of reasoning about incorrect inputs, as long as we are willing to accept that the program as a whole may crash, which is already a possibility in many *effectful* statically typed languages.

However, the dichotomy between gradual and optional typing is not as firm as one might expect. There have been many different proposed semantics of run-time type checking: "transient" cast semantics [87] only checks the head connective of a type (number, function, list, ...), "eager" cast semantics [39] checks run-time type information on closures, whereas "lazy" cast semantics [25] will always delay a type-check on a function until it is called (and there are other possibilities, see e.g. [31, 73]). The extent to which these different semantics have been shown to validate type-based reasoning has been limited to syntactic *gradual type soundness* and *blame soundness* theorems. In their strongest form, these theorems say: "If t is a closed program of type A then it diverges, or reduces to a runtime error blaming dynamically typed code, or reduces to a value that satisfies A to a certain extent." However, the theorem at this level of generality is quite weak, and justifies almost no program equivalences without more information. Saying that a resulting value satisfies type A might be a strong statement, but in transient semantics constrains only the head connective. The blame soundness theorem might also be quite strong, but depends on the definition of blame, which is part of the operational semantics of the language being defined.

We argue that existing gradual type soundness theorems are only indirectly expressing the actual desired properties of the gradual language, which are *program equivalences in the typed portion of the code* that are not valid in the dynamically typed portion. These typed equivalences are essential for ensuring that any reasoning about refactoring or optimization of code that is valid in a fully static setting is also valid for statically typed portions of a gradually typed program. Thus, preserving appropriate typed equivalences—the ones that justify refactoring and optimization—should be one of the criteria that gradually typed languages should satisfy.

So what are the program equivalences that hold in statically typed portions of the code but not in dynamically typed portions? They typically include β-like principles, which arise from computation steps, as well as η *equalities*, which express the uniqueness or universality of certain constructions.

The η law of the untyped λ-calculus, which states that any λ-term $M \equiv \lambda x.Mx$, is restricted in a typed language to only hold for terms of function type $M : A \to B$ (i.e., λ is the unique/universal way of making an element of the function type). This famously "fails" to hold in call-by-value languages in the presence of effects: if M is a program that prints `"hello"` before returning a function, then M will print *now*, whereas $\lambda x.Mx$ will only print when given an argument. But this can be accommodated with one further modification: the η law is valid in simple call-by-value languages[1] (e.g. SML) if we have a "value restriction" $V \equiv \lambda x.Vx$.

The above illustrates that η/extensionality rules must be stated for each type connective, and be sensitive to the effects/evaluation order of the terms involved. For instance, the η principle for the boolean type `Bool` *in call-by-value* is that for any term M with a free variable x : `Bool`, M is equivalent to a term that performs an if statement on x:

$$M \equiv \texttt{if}\ x\ \texttt{then}\ M[\texttt{true}/x]\ \texttt{else}\ M[\texttt{false}/x].$$

If we have an `if` form that is strongly typed (i.e., errors on non-booleans) then this tells us that it is *safe* to run an if statement on any input of boolean type (in CBN, by contrast an if statement forces a thunk and so is not necessarily safe). In addition, even if our `if` statement does some kind of coercion, this tells us that the term M only cares about whether x is "truthy" or "falsy" and so a client is free to change e.g. one truthy value to a different one without changing behavior.

This η principle justifies a number of program optimizations, such as dead-code and common subexpression elimination, and hoisting an if statement outside of the body of a function if it is well-scoped:

$$\lambda x.\texttt{if}\ y\ \texttt{then}\ M\ \texttt{else}\ N \equiv \texttt{if}\ y\ \texttt{then}\ \lambda x.M\ \texttt{else}\ \lambda x.N.$$

Any eager datatype, one whose elimination form is given by pattern matching such as $0, +, 1, \times, \texttt{list}$, has a similar η principle which enables similar reasoning, such as proofs by induction. The η principles for lazy types *in call-by-name* support dual behavioral reasoning about lazy functions, records, and streams.

1 This does not hold in languages with some intensional feature of functions such as reference equality. We discuss the applicability of our main results more generally in §5.4.

5.1.2 *An Axiomatic Approach to Gradual Typing*

In this paper, we systematically study questions of program equivalence for a class of gradually typed languages by working in an *axiomatic theory* of gradual program equivalence, a language and logic we call *gradual type theory* (GTT). Gradual type theory is the combination of a language of terms and gradual types with a simple logic for proving program equivalence and *error approximation* (equivalence up to one program erroring when the other does not) results. The logic axiomatizes the equational properties gradual programs should satisfy, and offers a high-level syntax for proving theorems about many languages at once: if a language models gradual type theory, then it satisfies all provable equivalences/approximations. Due to its type-theoretic design, different axioms of program equivalence are easily added or removed. The critical benefit of gradual type theory (GTT) is that it can be used both to explore language design questions and to verify behavioral properties of specific programs, such as correctness of optimizations and refactorings.

To get off the ground, we take two properties of the gradual language for granted. First, we assume a compositionality property: that any cast from A to B can be factored through the dynamic type ?, i.e., the cast $\langle B \Leftarrow A \rangle t$ is equivalent to first casting up from A to ? and then down to B: $\langle B \Leftarrow ? \rangle \langle ? \Leftarrow A \rangle t$. These casts often have quite different performance characteristics, but should have the same extensional behavior: of the cast semantics presented in Siek, Garcia, and Taha [73], only the partially eager detection strategy violates this principle, and this strategy is not common.

The second property we take for granted is that the language satisfies graduality, that if we change the types in a program to be more precise the program will either produce the same behavior as the original or raise a dynamic type error. Conversely, if a program does not error and some types are made "less precise" then behavior does not change.

Next, we study what program equivalences are provable in GTT under various assumptions. Our central application is to study when the β, η equalities are satisfied in a gradually typed language. We approach this problem by a surprising tack: rather than defining the behavior of dynamic type casts and then verifying or invalidating the β and η equalities, we *assume* the language satisfies β and η equality and then show that certain reductions of casts are in fact program equivalence *theorems* deducible from the axioms of GTT.

The cast reductions that we show satisfy all three constraints are those given by the "lazy cast semantics" [25, 73]. As a contrapositive, any gradually typed language for which these reductions are not program equivalences is *not* a model of the axioms of gradual type theory. This mean the language violates either compositionality, the

gradual guarantee, or one of the β, η axioms—and in practice, it is usually η.

For instance, a transient semantics, where only the top-level connectives are checked, violates η for strict pairs

$$x : A_1 \times A_2 \vdash (\texttt{let}\ (x_1, x_2) = x; 0) \not\equiv 0$$

because the top-level connectives of A_1 and A_2 are only checked when the pattern match is introduced. As a concrete counterexample to contextual equivalence, let A_1, A_2 all be `String`. Because only the top-level connective is checked, $(0, 1)$ is a valid value of type $\texttt{String} \times \texttt{String}$, but pattern matching on the pair ensures that the two components are checked to be strings, so the left-hand side raises a type error:

$$\texttt{let}\ (x_1, x_2) = (0, 1); 0 \mapsto \mathcal{O}.$$

On the right-hand side, with no pattern match, 0 is returned. This means simple program changes that are valid in a typed language, such as changing a function of two arguments to take a single pair of those arguments, are invalidated by the transient semantics. In summary, transient semantics is "lazier" than the types dictate, catching errors only when the term is inspected.

As a subtler example, in call-by-value "eager cast semantics" the $\beta\eta$ principles for all of the eager datatypes ($0, +, 1, \times$, lists, etc.) will be satisfied, but the η principle for the function type $\to$ is violated: there are values $V : A \to A'$ for which $V \neq \lambda x : A.Vx$. For instance, take an arbitrary function value $V : A \to \texttt{String}$ for some type A, and let $V' = \langle A \to ? \Leftarrow A \to \texttt{String} \rangle V$ be the result of casting it to have a dynamically typed output. Then in eager semantics, the following programs are not equivalent:

$$\lambda x : A.V'x \neq V' : A \to ?$$

We cannot observe any difference between these two programs by applying them to arguments, however, they are distinguished from each other by their behavior when *cast*. Specifically, if we cast both sides to $A \to \texttt{Number}$, then $\langle A \to \texttt{Number} \Leftarrow A \to ? \rangle (\lambda x : A.V'x)$ is a value, but $\langle A \to \texttt{Number} \Leftarrow A \to ? \rangle V'$ reduces to an error because `Number` is incompatible with `String`. However this type error might not correspond to any actual typing violation of the program involved. For one thing, the resulting function might never be executed. Furthermore, in the presence of effects, it may be that the original function $V : A \to \texttt{String}$ never returns a string (because it diverges, raises an exception or invokes a continuation), and so that same value casted to $A \to \texttt{Number}$ might be a perfectly valid inhabitant of that type. In summary the "eager" cast semantics is in fact overly eager: in its effort to find bugs faster than "lazy" semantics it disables the very type-based reasoning that gradual typing should provide.

While criticisms of transient semantics on the basis of type soundness have been made before [33], our development shows that the η principles of types are enough to *uniquely* determine a cast semantics, and helps clarify the trade-off between eager and lazy semantics of function casts.

5.1.3 *Technical Overview of GTT*

In this chapter, we develop an axiomatic gradual type theory GTT for a unified language that includes *both* call-by-value/eager types and call-by-name/lazy types (Sections 5.2, 5.3). Later in Chapter 7, we show that it is sound for contextual equivalence via a logical relations model. Because the η principles for types play a key role in our approach, it is necessary to work in a setting where we can have η principles for both eager and lazy types. We use Levy's Call-by-Push-Value [45] (CBPV), which fully and faithfully embeds both call-by-value and call-by-name evaluation with both eager and lazy datatypes,[2] and underlies much recent work on reasoning about effectful programs [8, 47]. GTT can prove results in and about existing call-by-value gradually typed languages, and also suggests a design for call-by-name and full call-by-push-value gradually typed languages.

A *type precision* relation $A \sqsubseteq A'$ (read: A is more precise than A', as in Siek et al. [75] and naïve subtyping [88]) controls which casts exist: a type precision $A \sqsubseteq A'$ induces an upcast from A to A' and a downcast from A' to A. Then, a *term precision* judgement is used for equational/approximational reasoning about programs. Term precision relates two terms whose types are related by type precision, and the upcasts and downcasts are each *specified* by certain term precision judgements holding. This specification axiomatizes only the properties of casts needed to ensure the graduality theorem, and not their precise behavior, so cast reductions can be *proved from it*, rather than stipulated in advance. The specification defines the casts "uniquely up to equivalence", which means that any two implementations satisfying it are behaviorally equivalent.

We generalize this axiomatic approach to call-by-push-value (§5.2), where there are both eager/value types and lazy/computation types. This is both a subtler question than it might at first seem, and has a surprisingly nice answer: we find that upcasts are naturally associated with eager/value types and downcasts with lazy/computation types, and that the modalities relating values and computations induce the downcasts for eager/value types and upcasts for lazy/computation types. Moreover, this analysis articulates an important behavioral property of casts that was proved operationally for call-by-value in [56] but missed for call-by-name in [58]: upcasts for eager types and

2 The distinction between "lazy" vs "eager" casts above is different than lazy vs. eager datatypes.

downcasts for lazy types are both "pure" in a suitable sense, which enables more refactorings and program optimizations. In particular, we show that these casts can be taken to be (and are essentially forced to be) "complex values" and "complex stacks" (respectively) in call-by-push-value, which corresponds to a behavioral property of *thunkability* and *linearity* [53]. We argue in §5.4 that this property is related to blame soundness.

Our gradual type theory naturally has two dynamic types, a dynamic eager/value type and a dynamic lazy/computation type, where the former can be thought of as a sum of all possible values, and the latter as a product of all possible behaviors. At the language design level, gradual type theory can be used to prove that, for a variety of eager/value and lazy/computation types, the "lazy" semantics of casts is the unique implementation satisfying β, η and graduality (§5.3). These behavioral equivalences can then be used in reasoning about optimizations, refactorings, and correctness of specific programs.

5.1.4 *Contributions.*

The main contributions of the chapter are as follows.

1. We present Gradual Type Theory in §5.2, a simple axiomatic theory of gradual typing. The theory axiomatizes three simple assumptions about a gradual language: compositionality, graduality, and type-based reasoning in the form of η equivalences.

2. We prove many theorems in the formal logic of Gradual Type Theory in §5.3. These include the unique implementation theorems for casts, which show that for each type connective of GTT, the η principle for the type ensures that the casts must implement the lazy contract semantics. Furthermore, we show that upcasts are always pure functions and dually that downcasts are always strict functions, as long as the base type casts are pure/strict.

In the following chapters, we will substantiate that our axiomatic theory is *reasonable* by (1) in Chapter 6 showing that GTT justifies surface language evaluation orders by elaborating them into GTT and (2) in Chapter 7 by constructing operational *models* of GTT that explicitly construct interpretations of the dynamic types as recursive types and give an operational semantics for which our axiomatic semantics is sound.

5.2 AXIOMATIC GRADUAL TYPE THEORY

In this section we introduce the syntax of Gradual Type Theory, an extension of Call-by-push-value [45] to support the constructions

$$
\begin{array}{rcl}
A & ::= & ? \mid U\underline{B} \mid 0 \mid A_1 + A_2 \mid 1 \mid A_1 \times A_2 \\
\underline{B} & ::= & ¿ \mid \underline{F}A \mid \top \mid \underline{B}_1 \,\&\, \underline{B}_2 \mid A \to \underline{B} \\
T & ::= & A \mid \underline{B} \\
\\
V & ::= & \langle A' \nwarrow A \rangle V \mid x \mid \mathtt{thunk}\ M \mid \mathtt{abort}\ V \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V \mid \mathtt{case}\ V\{x_1.V_1 \mid x_2.V_2\} \\
 & & \mid () \mid \mathtt{split}\ V\ \mathtt{to}\ ().V' \mid (V_1, V_2) \mid \mathtt{let}\ (x,y) = V; V' \\
\\
M, S & ::= & \langle \underline{B} \swarrow \underline{B}' \rangle M \mid \bullet \mid \mho_{\underline{B}} \mid \mathtt{force}\ V \mid \mathtt{abort}\ V \mid \mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \\
 & & \mid \mathtt{split}\ V\ \mathtt{to}\ ().M \mid \mathtt{let}\ (x,y) = V; M \\
 & & \mid \mathtt{ret}\ V \mid x \leftarrow M; N \mid \{\} \mid (M_1, M_2) \mid \pi M \mid \pi' M \mid \lambda x : A.M \mid M\,V \\
\\
E & ::= & V \mid M \\
\\
\Gamma & ::= & \cdot \mid \Gamma, x : A \\
\Delta & ::= & \cdot \mid \bullet : \underline{B} \\
\Phi & ::= & \cdot \mid \Phi, x \sqsubseteq x' : A \sqsubseteq A' \\
\Psi & ::= & \cdot \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'
\end{array}
$$

Figure 5.1: GTT Type and Term Syntax

of gradual typing. First we introduce call-by-push-value and then describe in turn the gradual typing features: dynamic types, casts, and the precision orderings on types and terms.

5.2.1 *Background: Call-by-Push-Value*

We present the syntax of GTT types and terms in Figure 5.1, and the typing rules in Figure 5.2. GTT is an extension of CBPV, so we first present CBPV as the unshaded rules in Figure 5.1. CBPV makes a distinction between *value types* A and *computation types* $\underline{B}$, where value types classify *values* $\Gamma \vdash V : A$ and computation types classify *computations* $\Gamma \vdash M : \underline{B}$. Effects are computations: for example, we might have an error computation $\mho_{\underline{B}} : \underline{B}$ of every computation type, or printing $\mathtt{print}\ V; M : \underline{B}$ if $V : \mathtt{string}$ and $M : \underline{B}$, which prints V and then behaves as M.

VALUE TYPES AND COMPLEX VALUES The value types include *eager* products 1 and $A_1 \times A_2$ and sums 0 and $A_1 + A_2$, which behave as in a call-by-value/eager language (e.g. a pair is only a value when its components are). The notion of value V is more permissive than one might expect, and expressions $\Gamma \vdash V : A$ are sometimes called *complex values* to emphasize this point: complex values include not

$$\boxed{\Gamma \vdash V : A \text{ and } \Gamma \mid \Delta \vdash M : \underline{B}}$$

$$\textsc{UpCast}\ \frac{\Gamma \vdash V : A \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A' \nwarrow A \rangle V : A'} \qquad \textsc{DnCast}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}' \qquad \underline{B} \sqsubseteq \underline{B}'}{\Gamma \mid \Delta \vdash \langle \underline{B} \swarrow \underline{B}' \rangle M : \underline{B}}$$

$$\textsc{Var}\ \frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \textsc{Hole}\ \frac{}{\Gamma \mid \bullet : \underline{B} \vdash \bullet : \underline{B}} \qquad \textsc{Err}\ \frac{}{\Gamma \mid \cdot \vdash \mathcal{O}_{\underline{B}} : \underline{B}}$$

$$U\text{I}\ \frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \vdash \mathtt{thunk}\ M : U\underline{B}} \qquad U\text{E}\ \frac{\Gamma \vdash V : U\underline{B}}{\Gamma \mid \cdot \vdash \mathtt{force}\ V : \underline{B}} \qquad F\text{I}\ \frac{\Gamma \vdash V : A}{\Gamma \mid \cdot \vdash \mathtt{ret}\ V : \underline{F}A}$$

$$F\text{E}\ \frac{\Gamma \mid \Delta \vdash M : \underline{F}A \qquad \Gamma, x : A \mid \cdot \vdash N : \underline{B}}{\Gamma \mid \Delta \vdash x \leftarrow M; N : \underline{B}}$$

$$0\text{E}\ \frac{\Gamma \vdash V : 0}{\Gamma \mid \Delta \vdash \mathtt{abort}\ V : T} \qquad +\text{I}_\text{L}\ \frac{\Gamma \vdash V : A_1}{\Gamma \vdash \mathtt{inl}\ V : A_1 + A_2}$$

$$+\text{I}_\text{R}\ \frac{\Gamma \vdash V : A_2}{\Gamma \vdash \mathtt{inr}\ V : A_1 + A_2} \qquad +\text{E}\ \frac{\Gamma \vdash V : A_1 + A_2 \quad \Gamma, x_1 : A_1 \mid \Delta \vdash E_1 : T \quad \Gamma, x_2 : A_2 \mid \Delta \vdash E_2 : T}{\Gamma \mid \Delta \vdash \mathtt{case}\ V\{x_1.E_1 \mid x_2.E_2\} : T}$$

$$1\text{I}\ \frac{}{\Gamma \vdash () : 1} \qquad 1\text{E}\ \frac{\Gamma \vdash V : 1 \qquad \Gamma \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \mathtt{split}\ V\ \mathtt{to}\ ().E : T}$$

$$\times\text{I}\ \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \times\text{E}\ \frac{\Gamma \vdash V : A_1 \times A_2 \quad \Gamma, x : A_1, y : A_2 \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \mathtt{let}\ (x, y)\ =\ V; E : T}$$

$$\to\text{I}\ \frac{\Gamma, x : A \mid \Delta \vdash M : \underline{B}}{\Gamma \mid \Delta \vdash \lambda x : A.M : A \to \underline{B}} \qquad \to\text{E}\ \frac{\Gamma \mid \Delta \vdash M : A \to \underline{B} \qquad \Gamma \vdash V : A}{\Gamma \mid \Delta \vdash M\,V : \underline{B}}$$

$$\top\text{I}\ \frac{}{\Gamma \mid \Delta \vdash \{\} : \top} \qquad \&\text{I}\ \frac{\Gamma \mid \Delta \vdash M_1 : \underline{B}_1 \qquad \Gamma \mid \Delta \vdash M_2 : \underline{B}_2}{\Gamma \mid \Delta \vdash (M_1, M_2) : \underline{B}_1 \,\&\, \underline{B}_2}$$

$$\&\text{E}\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \mid \Delta \vdash \pi M : \underline{B}_1} \qquad \&\text{E}'\ \frac{\Gamma \mid \Delta \vdash M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \mid \Delta \vdash \pi' M : \underline{B}_2}$$

Figure 5.2: GTT Typing

only closed runtime values, but also open values that have free value variables (e.g. $x : A_1, x_2 : A_2 \vdash (x_1, x_2) : A_1 \times A_2$), and expressions that pattern-match on values (e.g. $p : A_1 \times A_2 \vdash \texttt{let}\ (x_1, x_2) = p; (x_2, x_1) : A_2 \times A_1$). Thus, the complex values $x : A \vdash V : A'$ are a syntactic class of "pure functions" from A to A' (though there is no pure function *type* internalizing this judgement), which can be treated like values by a compiler because they have no effects (e.g. they can be dead-code-eliminated, common-subexpression-eliminated, and so on). In focusing [7] terminology, complex values consist of left inversion and right focus rules. For each pattern-matching construct (e.g. case analysis on a sum, splitting a pair), we have both an elimination rule whose branches are values (e.g. $\texttt{let}\ (x_1, x_2) = p; V$) and one whose branches are computations (e.g. $\texttt{let}\ (x_1, x_2) = p; M$). To abbreviate the typing rules for both in Figure 5.2, we use the following convention defined in Figure 5.1: E for either a complex value or a computation, and T for either a value type A or a computation type $\underline{B}$, and a judgement $\Gamma \mid \Delta \vdash E : T$ for either $\Gamma \vdash V : A$ or $\Gamma \mid \Delta \vdash M : \underline{B}$. Complex values can be translated away without loss of expressiveness by moving all pattern-matching into computations (see §7.3), at the expense of using a behavioral condition of *thunkability* [53] to capture the properties complex values have, such as being reordered, (de-)duplicated, etc.

this is a bit of an abuse of notation because Δ is not present in the former

SHIFTS A key notion in CBPV is the *shift* types $\underline{F}A$ and $U\underline{B}$, which mediate between value and computation types: $\underline{F}A$ is the computation type of potentially effectful programs that return a value of type A, while $U\underline{B}$ is the value type of thunked computations of type $\underline{B}$. The introduction rule for $\underline{F}A$ is returning a value of type A ($\texttt{ret}\ V$), while the elimination rule is sequencing a computation $M : \underline{F}A$ with a computation $x : A \vdash N : \underline{B}$ to produce a computation of a $\underline{B}$ ($x \leftarrow M; N$). While any closed complex value V is equivalent to an actual value, a computation of type $\underline{F}A$ might perform effects (e.g. printing) before returning a value, or might error or diverge and not return a value at all. The introduction and elimination rules for U are written $\texttt{thunk}\ M$ and $\texttt{force}\ V$, and say that computations of type $\underline{B}$ are bijective with values of type $U\underline{B}$. As an example of the action of the shifts, 0 is the empty value type, so $\underline{F}0$ classifies effectful computations that never return, but may perform effects (and then, must e.g. diverge or error), while $U\underline{F}0$ is the value type where such computations are thunked/delayed and considered as values. 1 is the trivial value type, so $\underline{F}1$ is the type of computations that can perform effects with the possibility of terminating successfully by returning $()$, and $U\underline{F}1$ is the value type where such computations are delayed values. $U\underline{F}$ is a monad on value types [51], while $\underline{F}U$ is a comonad on computation types.

COMPUTATION TYPES The computation type constructors in CBPV include first the lazy unit $\top$ and lazy product $\underline{B}_1 \,\&\, \underline{B}_2$, which behave as in a call-by-name language (e.g. a component of a lazy pair is evaluated only when it is projected). Functions $A \to \underline{B}$ have a value type as input and a computation type as a result. The equational theory of effects in CBPV computations may be surprising to those familiar only with call-by-value, because at higher computation types effects have a call-by-name-like equational theory. For example, at computation type $A \to \underline{B}$, we have an equality $\texttt{print}\ c; \lambda x.M = \lambda x.\texttt{print}\ c; M$. Intuitively, the reason is that $A \to \underline{B}$ is not treated as an *observable* type (one where computations are run): the states of the operational semantics are only those computations of type $\underline{F}A$ for some value type A. Thus, "running" a function computation means supplying it with an argument, and applying both of the above to an argument V is defined to result in $\texttt{print}\ c; M[V/x]$. This does *not* imply that the corresponding equations holds for the call-by-value function type, which we discuss below. As another example, *all* computations are considered equal at type $\top$, even computations that perform different effects ($\texttt{print}\ c$ vs. $\{\}$ vs. $\mho$), because there is by definition *no* way to extract an observable type $\underline{F}A$ from a computation of type $\top$. Consequently, $U\top$ is isomorphic to 1.

COMPLEX STACKS Just as the complex values V are a syntactic class terms that have no effects, CBPV includes a judgement for "stacks" S, a syntactic class of terms that reflect *all* effects of their input. A *stack* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{B}'$ can be thought of as a linear/strict function from $\underline{B}$ to $\underline{B}'$, which *must* use its input hole $\bullet$ *exactly* once at the head redex position. Consequently, effects can be hoisted out of stacks, because we know the stack will run them exactly once and first. For example, there will be contextual equivalences $S[\mho/\bullet] = \mho$ and $S[\texttt{print}\ V; M] = \texttt{print}\ V; S[M/\bullet]$. Just as complex values include pattern-matching, *complex stacks* include pattern-matching on values and introduction forms for the stack's output type. For example, $\bullet : \underline{B}_1 \,\&\, \underline{B}_2 \vdash (\pi'\bullet, \pi\bullet) : \underline{B}_2 \,\&\, \underline{B}_1$ is a complex stack, even though it mentions $\bullet$ more than once, because running it requires choosing a projection to get to an observable of type $\underline{F}A$, so *each time it is run* it uses $\bullet$ exactly once. In focusing terms, complex stacks include both left and right inversion, and left focus rules. In the equational theory of CBPV, $\underline{F}$ and U are *adjoint*, in the sense that stacks $\bullet : \underline{F}A \vdash S : \underline{B}$ are bijective with values $x : A \vdash V : U\underline{B}$, as both are bijective with computations $x : A \vdash M : \underline{B}$.

To compress the presentation in Figure 5.2, we use a typing judgement $\Gamma \mid \Delta \vdash M : \underline{B}$ with a "stoup", a typing context Δ that is either empty or contains exactly one assumption $\bullet : \underline{B}$, so $\Gamma \mid \cdot \vdash M : \underline{B}$ is a computation, while $\Gamma \mid \bullet : \underline{B} \vdash M : \underline{B}'$ is a stack. The typing rules for $\top$ and & treat the stoup additively (it is arbitrary in the conclusion

and the same in all premises); for a function application to be a stack, the stack input must occur in the function position. The elimination form for $U\underline{B}$, $\texttt{force}\ V$, is the prototypical non-stack computation (Δ is required to be empty), because forcing a thunk does not use the stack's input.

EMBEDDING CALL-BY-VALUE AND CALL-BY-NAME To translate from call-by-value (CBV) to CBPV, a CBV expression $x_1 : A_1, \ldots, x_n : A_n \vdash e : A$ is interpreted as a computation $x_1 : A_1^v, \ldots, x_n : A_n^v \vdash e^v : \underline{F}A^v$, where call-by-value products and sums are interpreted as $\times$ and $+$, and the call-by-value function type $A \to A'$ as $U(A^v \to \underline{F}A'^v)$. Thus, a call-by-value term $e : A \to A'$, which should mean an effectful computation of a function value, is translated to a computation $e^v : \underline{F}U(A^v \to \underline{F}A'^v)$. Here, the comonad $\underline{F}U$ offers an opportunity to perform effects *before* returning a function value—so under translation the CBV terms $\texttt{print}\ c; \lambda x.e$ and $\lambda x.\texttt{print}\ c; e$ will not be contextually equivalent. To translate call-by-name (CBN) to CBPV, a judgement $x_1 : \underline{B}_1, \ldots, x_m : \underline{B}_m \vdash e : \underline{B}$ is translated to $x_1 : U\underline{B_1}^n, \ldots, x_m : U\underline{B_m}^n \vdash e^n : \underline{B}^n$, representing the fact that call-by-name terms are passed thunked arguments. Product types are translated to $\top$ and $\times$, while a CBN function $B \to B'$ is translated to $U\underline{B}^n \to \underline{B'}^n$ with a thunked argument. Sums $B_1 + B_2$ are translated to $\underline{F}(U\underline{B_1}^n + U\underline{B_2}^n)$, making the "lifting" in lazy sums explicit. Call-by-push-value *subsumes* call-by-value and call-by-name in that these embeddings are *full and faithful*: two CBV or CBN programs are equivalent if and only if their embeddings into CBPV are equivalent, and every CBPV program with a CBV or CBN type can be back-translated.

EXTENSIONALITY/η PRINCIPLES The main advantage of CBPV for our purposes is that it accounts for the η/extensionality principles of both eager/value and lazy/computation types, because value types have η principles relating them to the value assumptions in the context Γ, while computation types have η principles relating them to the result type of a computation $\underline{B}$. For example, the η principle for sums says that any complex value or computation $x : A_1 + A_2 \vdash E : T$ is equivalent to $\texttt{case}\ x\{x_1.E[\texttt{inl}\ x_1/x] \mid x_2.E[\texttt{inr}\ x_2/x]\}$, i.e. a case on a value can be moved to any point in a program (where all variables are in scope) in an optimization. Given this, the above translations of CBV and CBN into CBPV explain why η for sums holds in CBV but not CBN: in CBV, $x : A_1 + A_2 \vdash E : T$ is translated to a term with $x : A_1 + A_2$ free, but in CBN, $x : B_1 + B_2 \vdash E : T$ is translated to a term with $x : U\underline{F}(U\underline{B}_1 + U\underline{B}_2)$ free, and the type $U\underline{F}(U\underline{B}_1 + U\underline{B}_2)$ of monadic computations that return a sum does not satisfy the η principle for sums in CBPV. Dually, the η principle for functions in CBPV is that any computation $M : A \to \underline{B}$ is equal to $\lambda x.M\,x$. A CBN term $e : B \to B'$ is translated to a CBPV computation of type $U\underline{B} \to \underline{B}'$,

to which CBPV function extensionality applies, while a CBV term $e : A \to A'$ is translated to a computation of type $\underline{F}U(A \to \underline{F}A')$, which does not satisfy the η rule for functions. We discuss a formal statement of these η principles with term precision below.

5.2.2 *Gradual Typing in GTT*

Next, we discuss the additions that make CBPV into our gradual type theory GTT.

THE DYNAMIC TYPE(S) A dynamic type plays a key role in gradual typing, and since GTT has two different kinds of types, we have a new question of whether the dynamic type should be a value type, or a computation type, or whether we should have *both* a dynamic value type and a dynamic computation type. Our modular, type-theoretic presentation of gradual typing allows us to easily explore these options, though we find that having both a dynamic value ? and a dynamic computation type $\underline{¿}$ gives the most natural implementation (see Chapter 7). Thus, we add both ? and $\underline{¿}$ to the grammar of types in Figure 5.1. We do *not* give introduction and elimination rules for the dynamic types, because we would like constructions in GTT to imply results for many different possible implementations of them. Instead, the terms for the dynamic types will arise from type precision and casts.

5.2.2.1 *Type Precision*

The *type precision* relation of gradual type theory is written $A \sqsubseteq A'$ and read as "A is more precise than A'"; intuitively, this means that A' supports more behaviors than A. Our previous work [56, 58] analyzes this as the existence of an *upcast* from A to A' and a downcast from A' to A which form an embedding-projection pair (*ep pair*) for term error approximation (an ordering where runtime errors are minimal): the upcast followed by the downcast is a no-op, while the downcast followed by the upcast might error more than the original term, because it imposes a run-time type check. Syntactically, type precision is defined (1) to be reflexive and transitive (a preorder), (2) where every type constructor is monotone in all positions, and (3) where the dynamic type is greatest in the type precision ordering. This last condition, *the dynamic type is the most dynamic type*, implies the existence of an upcast $\langle ? \nwarrow A \rangle$ and a downcast $\langle A \swarrow ? \rangle$ for every type A: any type can be embedded into it and projected from it. However, this by design does not characterize ? uniquely—instead, it is open-ended exactly which types exist (so that we can always add more), and some properties of the casts are undetermined; we will exploit this freedom in Chapter 7 when we explore some additional types and axioms.

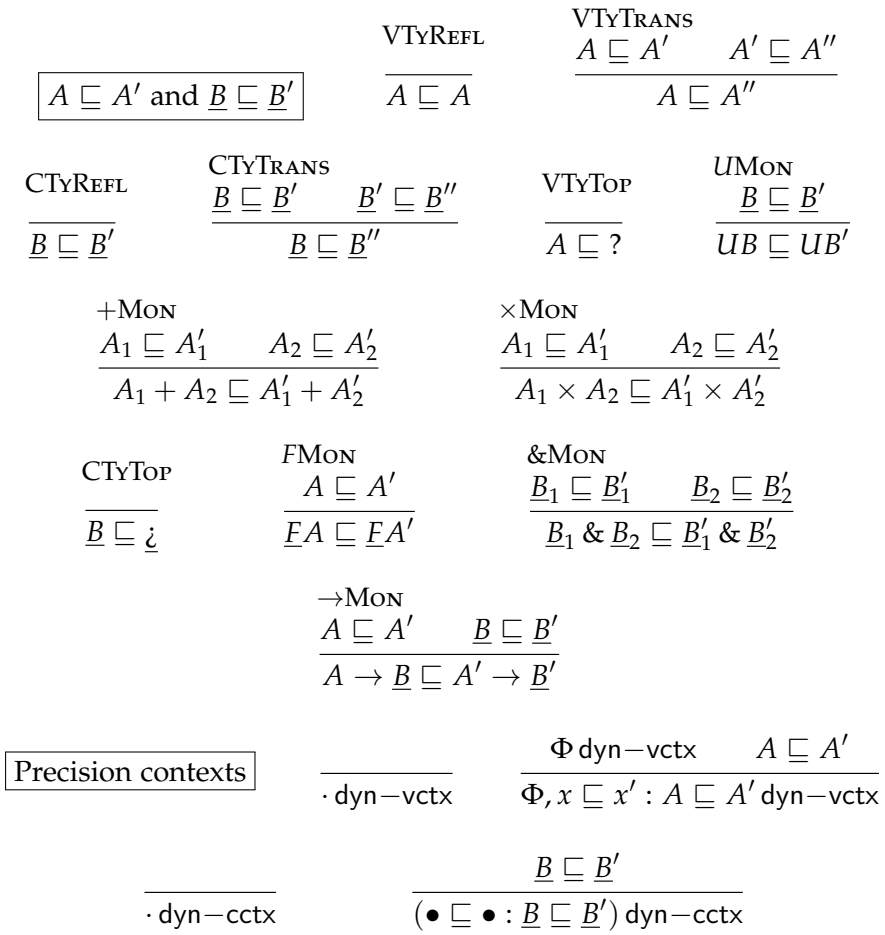

Figure 5.3: GTT Type Precision and Precision Contexts

This extends in a straightforward way to CBPV's distinction between value and computation types in Figure 5.3: there is a type precision relation for value types $A \sqsubseteq A'$ and for computation types $\underline{B} \sqsubseteq \underline{B}'$, which (1) each are preorders (VTyRefl, VTyTrans, CTyRefl, CTyTrans), (2) every type constructor is monotone (+Mon, ×Mon, &Mon ,→Mon) where the shifts $\underline{F}$ and U switch which relation is being considered (UMon, FMon), and (3) the dynamic types ? and $\underline{¿}$ are the most dynamic value and computation types respectively (VTyTop, CTyTop). For example, we have $U(A \to \underline{F}A') \sqsubseteq U(? \to \underline{F}?)$, which is the analogue of $A \to A' \sqsubseteq ? \to ?$ in call-by-value: because $\to$ preserves embedding-retraction pairs, it is monotone, not contravariant, in the domain [56, 58].

5.2.2.2 *Casts*

It is not immediately obvious how to add type casts to CBPV, because CBPV exposes finer judgemental distinctions than previous work considered. However, we can arrive at a first proposal by considering

how previous work would be embedded into CBPV. In the previous work on both CBV and CBN [56, 58] every type precision judgement $A \sqsubseteq A'$ induces both an upcast from A to A' and a downcast from A' to A. Because CBV types are associated to CBPV value types and CBN types are associated to CBPV computation types, this suggests that each value type precision $A \sqsubseteq A'$ should induce an upcast and a downcast, and each computation type precision $\underline{B} \sqsubseteq \underline{B}'$ should also induce an upcast and a downcast. In CBV, a cast from A to A' typically can be represented by a CBV function $A \to A'$, whose analogue in CBPV is $U(A \to \underline{F}A')$, and values of this type are bijective with computations $x : A \vdash M : \underline{F}A'$, and further with stacks $\bullet : \underline{F}A \vdash S : \underline{F}A'$. This suggests that a *value* type precision $A \sqsubseteq A'$ should induce an embedding-projection pair of *stacks* $\bullet : \underline{F}A \vdash S_u : \underline{F}A'$ and $\bullet : \underline{F}A' \vdash S_d : \underline{F}A$, which allow both the upcast and downcast to a priori be effectful computations. Dually, a CBN cast typically can be represented by a CBN function of type $B \to B'$, whose CBPV analogue is a computation of type $U\underline{B} \to \underline{B}'$, which is equivalent with a computation $x : U\underline{B} \vdash M : \underline{B}'$, and with a value $x : U\underline{B} \vdash V : U\underline{B}'$. This suggests that a *computation* type precision $\underline{B} \sqsubseteq \underline{B}'$ should induce an embedding-projection pair of *values* $x : U\underline{B} \vdash V_u : U\underline{B}'$ and $x : U\underline{B}' \vdash V_d : U\underline{B}$, where both the upcast and the downcast again may a priori be (co)effectful, in the sense that they may not reflect all effects of their input.

However, this analysis ignores an important property of CBV casts in practice: *upcasts* always terminate without performing any effects, and in some systems upcasts are even defined to be values, while only the *downcasts* are effectful (introduce errors). For example, for many types A, the upcast from A to ? is an injection into a sum/recursive type, which is a value constructor. Our previous work on a logical relation for call-by-value gradual typing [56] proved that all upcasts were pure in this sense as a consequence of the embedding-projection pair properties (but their proof depended on the only effects being divergence and type error). In GTT, we can make this property explicit in the syntax of the casts, by making the upcast $\langle A' \nwarrow A \rangle$ induced by a value type precision $A \sqsubseteq A'$ itself a complex value, rather than computation. On the other hand, many downcasts between value types are implemented as a case-analysis looking for a specific tag and erroring otherwise, and so are not complex values.

We can also make a dual observation about CBN casts. The *downcast* arising from $\underline{B} \sqsubseteq \underline{B}'$ has a stronger property than being a computation $x : U\underline{B}' \vdash M : \underline{B}$ as suggested above: it can be taken to be a stack $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$, because a downcasted computation evaluates the computation it is "wrapping" exactly once. One intuitive justification for this point of view, which we make precise in Chapter 7, is to think of the dynamic computation type $\underline{?}$ as a recursive *product* of all possible behaviors that a computation might have, and the downcast

as a recursive type unrolling and product projection, which is a stack. From this point of view, an *upcast* can introduce errors, because the upcast of an object supporting some "methods" to one with all possible methods will error dynamically on the unimplemented ones.

These observations are expressed in the (shaded) UpCast and DnCasts rules for casts in Figure 5.2: the upcast for a value type precision is a complex value, while the downcast for a computation type precision is a stack (if its argument is). Indeed, this description of casts is simpler than the intuition we began the section with: rather than putting in both upcasts and downcasts for all value and computation type precisions, it suffices to put in only *upcasts* for *value* type precisions and *downcasts* for *computation* type precisions, because of monotonicity of type precision for $U/\underline{F}$ types. The *downcast* for a *value* type precision $A \sqsubseteq A'$, as a stack $\bullet : \underline{F}A' \vdash \langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet : \underline{F}A$ as described above, is obtained from $\underline{F}A \sqsubseteq \underline{F}A'$ as computation types. The upcast for a computation type precision $\underline{B} \sqsubseteq \underline{B}'$ as a value $x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle x : U\underline{B}'$ is obtained from $U\underline{B} \sqsubseteq U\underline{B}'$ as value types. Moreover, we will show below that the value upcast $\langle A' \nwarrow A \rangle$ induces a stack $\bullet : \underline{F}A \vdash \ldots : \underline{F}A'$ that behaves like an upcast, and dually for the downcast, so this formulation implies the original formulation above.

We justify this design in two ways in the remainder of the paper. In Chapter 7, we show how to implement casts by a contract translation to CBPV where upcasts are complex values and downcasts are complex stacks. However, one goal of GTT is to be able to prove things about many gradually typed languages at once, by giving different models, so one might wonder whether this design rules out useful models of gradual typing where casts can have more general effects. In Theorem 85, we show instead that our design choice is forced for all casts, as long as the casts between ground types and the dynamic types are values/stacks.

5.2.2.3 *Term Precision: Judgements and Structural Rules*

The final piece of GTT is the *term precision* relation, a syntactic judgement that is used for reasoning about the behavioral properties of terms in GTT. To a first approximation, term precision can be thought of as syntactic rules for reasoning about *contextual approximation* relative to errors (not divergence), where $E \sqsubseteq E'$ means that either E errors or E and E' have the same result. However, a key idea in GTT is to consider a *heterogeneous* term precision judgement $E \sqsubseteq E' : T \sqsubseteq T'$ between terms $E : T$ and $E' : T'$ where $T \sqsubseteq T'$—i.e. relating two terms at two different types, where the type on the right is less precise than the type on the right. This judgement structure allows simple axioms characterizing the behavior of casts [58] and axiomatizes the graduality property [75]. Here, we break this judgement up into value precision $V \sqsubseteq V' : A \sqsubseteq A'$ and computation precision $M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$. To

$$\boxed{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \text{ and } \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynRefl}\ \frac{}{\Gamma \sqsubseteq \Gamma \mid \Delta \sqsubseteq \Delta \vdash E \sqsubseteq E : T \sqsubseteq T} \qquad \textsc{TmDynVar}\ \frac{}{\Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \vdash x \sqsubseteq x' : A \sqsubseteq A'}$$

$$\textsc{TmDynTrans}\ \frac{\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash E \sqsubseteq E' : T \sqsubseteq T' \quad \Gamma' \sqsubseteq \Gamma'' \mid \Delta' \sqsubseteq \Delta'' \vdash E' \sqsubseteq E'' : T' \sqsubseteq T''}{\Gamma \sqsubseteq \Gamma'' \mid \Delta \sqsubseteq \Delta'' \vdash E \sqsubseteq E'' : T \sqsubseteq T''}$$

$$\textsc{TmDynValSubst}\ \frac{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \quad \Phi, x \sqsubseteq x' : A \sqsubseteq A', \Phi' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'}{\Phi \mid \Psi \vdash E[V/x] \sqsubseteq E'[V'/x'] : T \sqsubseteq T'}$$

$$\textsc{TmDynHole}\ \frac{}{\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'}$$

$$\textsc{TmDynStkSubst}\ \frac{\Phi \mid \Psi \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \sqsubseteq \underline{B}_1' \quad \Phi \mid \bullet \sqsubseteq \bullet : \underline{B}_1 \sqsubseteq \underline{B}_1' \vdash M_2 \sqsubseteq M_2' : \underline{B}_2 \sqsubseteq \underline{B}_2'}{\Phi \mid \Psi \vdash M_2[M_1/\bullet] \sqsubseteq M_2'[M_1'/\bullet] : \underline{B}_2 \sqsubseteq \underline{B}_2'}$$

Figure 5.4: GTT Term Precision (Structural and Congruence Rules)

support reasoning about open terms, the full form of the judgements are

- $\Gamma \sqsubseteq \Gamma' \vdash V \sqsubseteq V' : A \sqsubseteq A'$ where $\Gamma \vdash V : A$ and $\Gamma' \vdash V' : A'$ and $\Gamma \sqsubseteq \Gamma'$ and $A \sqsubseteq A'$.
- $\Gamma \sqsubseteq \Gamma' \mid \Delta \sqsubseteq \Delta' \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'$ where $\Gamma \mid \Delta \vdash M : \underline{B}$ and $\Gamma' \mid \Delta' \vdash M' : \underline{B}'$.

where $\Gamma \sqsubseteq \Gamma'$ is the pointwise lifting of value type precision, and $\Delta \sqsubseteq \Delta'$ is the analogous lifting of computation type precision. We write $\Phi : \Gamma \sqsubseteq \Gamma'$ and $\Psi : \Delta \sqsubseteq \Delta'$ as syntax for "zipped" pairs of contexts that are pointwise related by type precision, $x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1', \ldots, x_n \sqsubseteq x_n' : A_n \sqsubseteq A_n'$, which correctly suggests that one can substitute related terms for related variables. We will implicitly zip/unzip pairs of contexts, and sometimes write e.g. $\Gamma \sqsubseteq \Gamma$ to mean $x \sqsubseteq x : A \sqsubseteq A$ for all $x : A$ in Γ.

The main point of our rules for term precision is that *there are no type-specific axioms in the definition* beyond the $\beta\eta$-axioms that the type satisfies in a non-gradual language. Thus, adding a new type to gradual type theory does not require any a priori consideration of its gradual behavior in the language definition; instead, this is deduced as a theorem in the type theory. The basic structural rules of term precision in Figure 5.4 and Figure 5.5 say that it is reflexive and transitive (TmDynRefl, TmDynTrans), that assumptions can be used and substituted for (TmDynVar, TmDynValSubst, TmDynHole, TmDynStkSubst), and that every term constructor is monotone (the Cong rules). While we could add congruence rules for errors and casts, these follow from the axioms characterizing their behavior below.

$$+\text{IlCong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A_1 \sqsubseteq A_1'}{\Phi \vdash \mathtt{inl}\ V \sqsubseteq \mathtt{inl}\ V' : A_1 + A_2 \sqsubseteq A_1' + A_2'}$$

$$+\text{IrCong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A_2 \sqsubseteq A_2'}{\Phi \vdash \mathtt{inr}\ V \sqsubseteq \mathtt{inr}\ V' : A_1 + A_2 \sqsubseteq A_1' + A_2'}$$

$$+\text{ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : A_1 + A_2 \sqsubseteq A_1' + A_2' \\ \Phi, x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1' \mid \Psi \vdash E_1 \sqsubseteq E_1' : T \sqsubseteq T' \\ \Phi, x_2 \sqsubseteq x_2' : A_2 \sqsubseteq A_2' \mid \Psi \vdash E_2 \sqsubseteq E_2' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \mathtt{case}\ V\{x_1.E_1 \mid x_2.E_2\} \sqsubseteq \mathtt{case}\ V\{x_1'.E_1' \mid x_2'.E_2'\} : T'}$$

$$0\text{ECong}\quad \frac{\Phi \vdash V \sqsubseteq V' : 0 \sqsubseteq 0}{\Phi \mid \Psi \vdash \mathtt{abort}\ V \sqsubseteq \mathtt{abort}\ V' : T \sqsubseteq T'} \qquad 1\text{ICong}\quad \frac{}{\Phi \vdash () \sqsubseteq () : 1 \sqsubseteq 1}$$

$$1\text{ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : 1 \sqsubseteq 1 \\ \Phi \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \mathtt{split}\ V\ \mathtt{to}\ ().E \sqsubseteq \mathtt{split}\ V\ \mathtt{to}\ ().'E' : T \sqsubseteq T'}$$

$$\times\text{ICong}\quad \frac{\begin{array}{c}\Phi \vdash V_1 \sqsubseteq V_1' : A_1 \sqsubseteq A_1' \\ \Phi \vdash V_2 \sqsubseteq V_2' : A_2 \sqsubseteq A_2'\end{array}}{\Phi \vdash (V_1, V_2) \sqsubseteq (V_1', V_2') : A_1 \times A_2 \sqsubseteq A_1' \times A_2'} \qquad \rightarrow\text{ICong}\quad \frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \Psi \vdash \lambda x : A.M \sqsubseteq \lambda x' : A'.M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

$$\times\text{ECong}\quad \frac{\begin{array}{c}\Phi \vdash V \sqsubseteq V' : A_1 \times A_2 \sqsubseteq A_1' \times A_2' \\ \Phi, x \sqsubseteq x' : A_1 \sqsubseteq A_1', y \sqsubseteq y' : A_2 \sqsubseteq A_2' \mid \Psi \vdash E \sqsubseteq E' : T \sqsubseteq T'\end{array}}{\Phi \mid \Psi \vdash \mathtt{let}\ (x,y) = V; E \sqsubseteq \mathtt{let}\ (x',y') = V'; E' : T \sqsubseteq T'}$$

$$\rightarrow\text{ECong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : A \to \underline{B} \sqsubseteq A' \to \underline{B}' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \Psi \vdash M\,V \sqsubseteq M'\,V' : \underline{B} \sqsubseteq \underline{B}'}$$

$$U\text{ICong}\quad \frac{\Phi \mid \cdot \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \vdash \mathtt{thunk}\ M \sqsubseteq \mathtt{thunk}\ M' : U\underline{B} \sqsubseteq U\underline{B}'} \qquad U\text{ECong}\quad \frac{\Phi \vdash V \sqsubseteq V' : U\underline{B} \sqsubseteq U\underline{B}'}{\Phi \mid \cdot \vdash \mathtt{force}\ V \sqsubseteq \mathtt{force}\ V' : \underline{B} \sqsubseteq \underline{B}'}$$

$$F\text{ICong}\quad \frac{\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \mid \cdot \vdash \mathtt{ret}\ V \sqsubseteq \mathtt{ret}\ V' : \underline{F}A \sqsubseteq \underline{F}A'}$$

$$F\text{ECong}\quad \frac{\begin{array}{c}\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{F}A \sqsubseteq \underline{F}A' \\ \Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \cdot \vdash N \sqsubseteq N' : \underline{B} \sqsubseteq \underline{B}'\end{array}}{\Phi \mid \Psi \vdash x \leftarrow M; N \sqsubseteq x' \leftarrow M'; N' : \underline{B} \sqsubseteq \underline{B}'} \qquad \top\text{ICong}\quad \frac{}{\Phi \mid \Psi \vdash \{\} \sqsubseteq \{\} : \top \sqsubseteq \top}$$

$$\&\text{ICong}\quad \frac{\Phi \mid \Psi \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \sqsubseteq \underline{B}_1' \qquad \Phi \mid \Psi \vdash M_2 \sqsubseteq M_2' : \underline{B}_2 \sqsubseteq \underline{B}_2'}{\Phi \mid \Psi \vdash (M_1, M_2) \sqsubseteq (M_1', M_2') : \underline{B}_1 \,\&\, \underline{B}_2 \sqsubseteq \underline{B}_1' \,\&\, \underline{B}_2'}$$

$$\&\text{ECong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B}_1 \,\&\, \underline{B}_2 \sqsubseteq \underline{B}_1' \,\&\, \underline{B}_2'}{\Phi \mid \Psi \vdash \pi M \sqsubseteq \pi M' : \underline{B}_1 \sqsubseteq \underline{B}_1'} \qquad \&\text{E}'\text{Cong}\quad \frac{\Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B}_1 \,\&\, \underline{B}_2 \sqsubseteq \underline{B}_1' \,\&\, \underline{B}_2'}{\Phi \mid \Psi \vdash \pi' M \sqsubseteq \pi' M' : \underline{B}_2 \sqsubseteq \underline{B}_2'}$$

Figure 5.5: GTT Term Precision (Congruence Rules)

We will often abbreviate a "homogeneous" term precision (where the type or context precision is given by reflexivity) by writing e.g. $\Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$ for $\Gamma \sqsubseteq \Gamma \vdash V \sqsubseteq V' : A \sqsubseteq A'$, or $\Phi \vdash V \sqsubseteq V' : A$ for $\Phi \vdash V \sqsubseteq V' : A \sqsubseteq A$, and similarly for computations. The entirely homogeneous judgements $\Gamma \vdash V \sqsubseteq V' : A$ and $\Gamma \mid \Delta \vdash M \sqsubseteq M' : \underline{B}$ can be thought of as a syntax for contextual error approximation (as we prove below). We write $V \sqsupseteq\sqsubseteq V'$ ("equiprecision") to mean term precision relations in both directions (which requires that the types are also equiprecise $\Gamma \sqsupseteq\sqsubseteq \Gamma'$ and $A \sqsubseteq A'$), which is a syntactic judgement for contextual equivalence.

5.2.2.4 *Term Precision Axioms*

Finally, we assert some term precision axioms that describe the behavior of programs. The cast universal properties at the top of Figure 5.6, following New and Licata [58], say that the defining property of an upcast from A to A' is that it is the most precise term of type A' that is less precise that x, a "least upper bound". That is, $\langle A' \nwarrow A \rangle x$ is a term of type A' that is less precise that x (the "bound" rule), and for any other term x' of type A' that is less precise than x, $\langle A' \nwarrow A \rangle x$ is more precise than x' (the "best" rule). Dually, the downcast $\langle \underline{B} \swarrow \underline{B}' \rangle \bullet$ is the most dynamic term of type $\underline{B}$ that is more precise than $\bullet$, a "greatest lower bound". These defining properties are entirely independent of the types involved in the casts, and do not change as we add or remove types from the system.

We will show that these defining properties already imply that the shift of the upcast $\langle A' \nwarrow A \rangle$ forms a Galois connection/adjunction with the downcast $\langle \underline{F}A \swarrow \underline{F}A' \rangle$, and dually for computation types (see Theorem 73). They do not automatically form a Galois insertion/coreflection/embedding-projection pair, but we can add this by the retract axioms in Figure 5.6. Together with other theorems of GTT, these axioms imply that any upcast followed by its corresponding downcast is the identity (see Theorem 74).

This specification of casts leaves some behavior undefined: for example, we cannot prove in the theory that $\langle \underline{F}1 + 1 \swarrow \underline{F}? \rangle \langle ? \nwarrow 1 \rangle$ reduces to an error. We choose this design because there are valid models in which it is not an error, for instance if the unique value of 1 is represented as the boolean `true`. In Chapter 7, we show additional axioms that fully characterize the behavior of the dynamic type.

The type universal properties in the middle of the figure, which are taken directly from CBPV, assert the $\beta\eta$ rules for each type as (homogeneous) term equiprecisions—these should be understood as having, as implicit premises, the typing conditions that make both sides type check, in equiprecise contexts.

The final axioms assert properties of the run-time error term $\mho$: it is the most precise term (has the fewest behaviors) of every computation type, and all complex stacks are strict in errors, because stacks force

their evaluation position. We state the first axiom in a heterogeneous way, which includes congruence $\Gamma \sqsubseteq \Gamma' \vdash \mho_{\underline{B}} \sqsubseteq \mho_{\underline{B}'} : \underline{B} \sqsubseteq \underline{B}'$.

5.3 THEOREMS IN GRADUAL TYPE THEORY

In this section, we show that the axiomatics of gradual type theory determine most properties of casts, which shows that these behaviors of casts are forced in any implementation of gradual typing satisfying graduality and β, η.

5.3.1 *Derived Cast Rules*

As noted above, monotonicity of type precision for U and $\underline{F}$ means that we have the following as instances of the general cast rules:

Lemma 66 (Shifted Casts)**.** *The following are derivable:*

$$\frac{\Gamma \mid \Delta \vdash M : \underline{F}A' \qquad A \sqsubseteq A'}{\Gamma \mid \Delta \vdash \langle \underline{F}A \swarrow \underline{F}A' \rangle M : \underline{F}A} \qquad \frac{\Gamma \vdash V : U\underline{B} \qquad \underline{B} \sqsubseteq \underline{B}'}{\Gamma \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle V : U\underline{B}'}$$

Proof. They are instances of the general upcast and downcast rules, using the fact that U and $\underline{F}$ are congruences for type precision, so in the first rule $\underline{F}A \sqsubseteq \underline{F}A'$, and in the second, $U\underline{B} \sqsubseteq U\underline{B}'$. □

The cast universal properties in Figure 5.6 imply the following seemingly more general rules for reasoning about casts:

Lemma 67 (Upcast and downcast left and right rules)**.** *The following are derivable:*

$$\frac{A \sqsubseteq A' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \vdash V \sqsubseteq \langle A'' \nwarrow A' \rangle V' : A \sqsubseteq A''} \text{ UpR}$$

$$\frac{\Phi \vdash V \sqsubseteq V'' : A \sqsubseteq A''}{\Phi \vdash \langle A' \nwarrow A \rangle V \sqsubseteq V'' : A' \sqsubseteq A''} \text{ UpL}$$

$$\frac{\underline{B}' \sqsubseteq \underline{B}'' \qquad \Phi \mid \Psi \vdash M' \sqsubseteq M'' : \underline{B}' \sqsubseteq \underline{B}''}{\Phi \mid \Psi \vdash \langle \underline{B} \swarrow \underline{B}' \rangle M' \sqsubseteq M'' : \underline{B} \sqsubseteq \underline{B}''} \text{ DnL}$$

$$\frac{\Phi \mid \Psi \vdash M \sqsubseteq M'' : B \sqsubseteq B''}{\Phi \mid \Psi \vdash M \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle M'' : \underline{B} \sqsubseteq \underline{B}''} \text{ DnR}$$

Proof. For upcast left, substitute V' into the axiom $x \sqsubseteq \langle A'' \nwarrow A' \rangle x : A' \sqsubseteq A''$ to get $V' \sqsubseteq \langle A'' \nwarrow A' \rangle V'$, and then use transitivity with the premise.

For upcast right, by transitivity of

$$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x' : A' \sqsubseteq A' \qquad x' \sqsubseteq x'' : A' \sqsubseteq A'' \vdash x' \sqsubseteq x'' : A' \sqsubseteq A''$$

Cast Universal Properties

	Bound	Best
Up	$x : A \vdash x \sqsubseteq \langle A' \nwarrow A \rangle x : A \sqsubseteq A'$	$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x' : A'$
Down	$\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$	$\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$

Retract Axiom

$$x : A \vdash \langle \underline{F}A \swarrow \underline{F}? \rangle(\texttt{ret}\ (\langle ? \nwarrow A \rangle x)) \sqsubseteq \texttt{ret}\ x : \underline{F}A$$
$$x : U\underline{B} \vdash \langle \underline{B} \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B} \rangle x)) \sqsubseteq \texttt{force}\ x : \underline{B}$$

Type Universal Properties

Type	β	η
$+$	$\texttt{case inl}\ V\{x_1.E_1 \mid \ldots\} \sqsupseteq\sqsubseteq E_1[V/x_1]$ $\texttt{case inr}\ V\{\ldots \mid x_2.E_2\} \sqsupseteq\sqsubseteq E_2[V/x_2]$	$E \sqsupseteq\sqsubseteq \texttt{case}\ x\{x_1.E[\texttt{inl}\ x_1/x]$ $\mid x_2.E[\texttt{inr}\ x_2/x]\}$ where $x : A_1 + A_2 \vdash E : T$
0	–	$E \sqsupseteq\sqsubseteq \texttt{abort}\ x$ where $x : 0 \vdash E : T$
$\times$	$\texttt{let}\ (x_1, x_2) = (V_1, V_2); E \sqsupseteq\sqsubseteq E[V_1/x_1, V_2/x_2]$	$E \sqsupseteq\sqsubseteq \texttt{let}\ (x_1, x_2) = x; E[(x_1, x_2)/x]$ where $x : A_1 \times A_2 \vdash E : T$
1	$\texttt{split}\ ()\ \texttt{to}\ ().E \sqsupseteq\sqsubseteq E$	$x : 1 \vdash E \sqsupseteq\sqsubseteq \texttt{split}\ x\ \texttt{to}\ ().E[()/x] : T$ where $x : 1 \vdash E : T$
U	$\texttt{force thunk}\ M \sqsupseteq\sqsubseteq M$	$x : U\underline{B} \vdash x \sqsupseteq\sqsubseteq \texttt{thunk force}\ x : U\underline{B}$
F	$x \leftarrow \texttt{ret}\ V; M \sqsupseteq\sqsubseteq M[V/x]$	$\bullet : \underline{F}A \vdash M \sqsupseteq\sqsubseteq x \leftarrow \bullet; M[\texttt{ret}\ x/\bullet]$
$\to$	$(\lambda x : A.M)\ V \sqsupseteq\sqsubseteq M[V/x]$	$\bullet : A \to \underline{B} \vdash \bullet \sqsupseteq\sqsubseteq \lambda x : A.\ \bullet\ x : A \to \underline{B}$
$\&$	$\pi(M, M') \sqsupseteq\sqsubseteq M$ $\pi'(M, M') \sqsupseteq\sqsubseteq M'$	$\bullet : \underline{B}_1\ \&\ \underline{B}_2 \vdash \bullet \sqsupseteq\sqsubseteq (\pi\bullet, \pi'\bullet)$
$\top$	-	$\bullet : \top \vdash \bullet \sqsupseteq\sqsubseteq \{\}$

Error Properties

$$\textsc{ErrBot}\ \frac{\Gamma' \mid \cdot \vdash M' : \underline{B}'}{\Gamma \sqsubseteq \Gamma' \mid \cdot \vdash \mho \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'} \qquad \textsc{StkStrict}\ \frac{\Gamma \mid x : \underline{B} \vdash S : \underline{B}'}{\Gamma \mid \cdot \vdash S[\mho_{\underline{B}}] \sqsubseteq \mho_{\underline{B}'} : \underline{B}'}$$

Figure 5.6: GTT Term Precision Axioms

we have

$$x \sqsubseteq x'' : A \sqsubseteq A'' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x'' : A' \sqsubseteq A''$$

Substituting the premise into this gives the conclusion.

For downcast left, substituting M' into the axiom $\langle \underline{B} \swarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$ gives $\langle \underline{B} \swarrow \underline{B}' \rangle M \sqsubseteq M$, and then transitivity with the premise gives the result.

For downcast right, transitivity of

$$\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \quad \bullet' \sqsubseteq \bullet'' : \underline{B}' \sqsubseteq \underline{B}'' \vdash \bullet' \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle \bullet''$$

gives $\bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}'' \vdash \bullet \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle \bullet''$, and then substitution of the premise into this gives the conclusion. □

In sequent calculus terminology, in the term precision judgement an upcast is left-invertible, while a downcast is right-invertible, in the sense that any time we have a conclusion with a upcast on the left/downcast on the right, we can without loss of generality apply these rules (this comes from upcasts and downcasts forming a Galois connection). We write the $A \sqsubseteq A'$ and $\underline{B}' \sqsubseteq \underline{B}''$ premises on the non-invertible rules to emphasize that the premise is not necessarily well-formed given that the conclusion is.

We did not include explicit congruence rules for casts in Figure 5.5 because they are derivable:

Lemma 68 (Cast congruence rules)**.** *The following congruence rules for casts are derivable:*

$$\frac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A'' \nwarrow A \rangle x \sqsubseteq \langle A'' \nwarrow A' \rangle x' : A''}$$

$$\frac{A \sqsubseteq A' \qquad A' \sqsubseteq A''}{x : A \vdash \langle A' \nwarrow A \rangle x \sqsubseteq \langle A'' \nwarrow A \rangle x : A' \sqsubseteq A''}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}' \qquad \underline{B}' \sqsubseteq \underline{B}''}{\bullet' \sqsubseteq \bullet'' : \underline{B}' \sqsubseteq \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet' \sqsubseteq \langle \underline{B} \swarrow \underline{B}'' \rangle \bullet'' : \underline{B}}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}' \qquad \underline{B}' \sqsubseteq \underline{B}''}{\bullet'' : \underline{B}'' \vdash \langle \underline{B} \swarrow \underline{B}'' \rangle \bullet'' \sqsubseteq \langle \underline{B}' \swarrow \underline{B}'' \rangle \bullet'' : \underline{B} \sqsubseteq \underline{B}'}$$

Proof. In all cases, uses the invertible and then non-invertible rule for the cast. For the first rule, by upcast left, it suffices to show $x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle A'' \nwarrow A' \rangle x' : A \sqsubseteq A''$ which is true by upcast right, using $x \sqsubseteq x'$ in the premise. The other cases follow by a similar argument. □

Next, while in GTT we assume the existence of upcast values from value precision and downcast stacks from computation precision,

sometimes we can prove that certain terms satisfy the following definition of "downcast value" and "upcast stack".

In GTT, we assert the existence of value upcasts and computation downcasts for derivable type precision relations. While we do not assert the existence of all *value* downcasts and *computation* upcasts, we can define the universal property that identifies a term as such:

Definition 69 (Upcast stack/Value downcast).

1. If $\underline{B} \sqsubseteq \underline{B}'$, a *stack upcast from B to B'* is a stack $\bullet : \underline{B} \vdash \langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle \bullet : \underline{B}'$ that satisfies the computation precision rules of an upcast $\bullet : \underline{B} \vdash \bullet \sqsubseteq \langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle \bullet : \underline{B} \sqsubseteq \underline{B}'$ and $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \langle\!\langle \underline{B}' \nwarrow \underline{B} \rangle\!\rangle \bullet \sqsubseteq \bullet' : \underline{B}'$.

2. If $A \sqsubseteq A'$, a *value downcast from A' to A* is a complex value $x : A' \vdash \langle\langle A \swarrow A' \rangle\rangle x : A$ that satisfies the value precision rules of a downcast $x : A' \vdash \langle\!\langle A \swarrow A' \rangle\!\rangle x \sqsubseteq x : A \sqsubseteq A'$ and $x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle\!\langle A \swarrow A' \rangle\!\rangle x' : A$.

One convenient application of this is that we can simplify the statement of several properties by "forgetting" that an upcast $\langle A' \nwarrow A \rangle$ is a value, and instead using a derivable upcast $\langle\!\langle \underline{F}A' \nwarrow \underline{F}A \rangle\!\rangle$ as defined in the following (and dually for computation types)

Definition 70 (Upcast stacks/Downcast values). If $A \sqsubseteq A'$, then we define

$$\langle\!\langle \underline{F}A' \nwarrow \underline{F}A \rangle\!\rangle E = x \leftarrow E; \mathtt{ret}\ \langle A' \nwarrow A \rangle x.$$

which is an upcast stack.
If $\underline{B} \sqsubseteq \underline{B}'$ then we define

$$\langle\!\langle U\underline{B} \swarrow U\underline{B}' \rangle\!\rangle V = \mathtt{thunk}\ (\langle \underline{B} \swarrow \underline{B}' \rangle (\mathtt{force}\ V))$$

which is a downcast value.

5.3.2 *Type-Generic Properties of Casts*

The universal property axioms for upcasts and downcasts in Figure 5.6 define them *uniquely* up to equiprecision ($\sqsupseteq\sqsubseteq$): anything with the same property is behaviorally equivalent to a cast.

Theorem 71 (Specification for Casts is a Universal Property).

1. *If $A \sqsubseteq A'$ and $x : A \vdash V : A'$ is a complex value such that $x : A \vdash x \sqsubseteq V : A \sqsubseteq A'$ and $x \sqsubseteq x' : A \sqsubseteq A' \vdash V \sqsubseteq x' : A'$ then $x : A \vdash V \sqsupseteq\sqsubseteq \langle A' \nwarrow A \rangle x : A'$.*

2. *If $\underline{B} \sqsubseteq \underline{B}'$ and $\bullet' : \underline{B}' \vdash S : \underline{B}$ is a complex stack such that $\bullet' : \underline{B}' \vdash S \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}'$ and $\bullet \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq S : \underline{B}$ then $\bullet' : \underline{B}' \vdash S \sqsupseteq\sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle \bullet' : \underline{B}$*

Proof. For the first part, to show $\langle A' \leftharpoonup A \rangle x \sqsubseteq V$, by upcast left, it suffices to show $x \sqsubseteq V : A \sqsubseteq A'$, which is one assumption. To show $V \sqsubseteq \langle A' \leftharpoonup A \rangle x$, we substitute into the second assumption with $x \sqsubseteq \langle A' \leftharpoonup A \rangle x : A \sqsubseteq A'$, which is true by upcast right.

For the second part, to show $S \sqsubseteq \langle \underline{B} \upharpoonleft \underline{B}' \rangle \bullet'$, by downcast right, it suffices to show $S \sqsubseteq \bullet' : \underline{B} \sqsubseteq \underline{B}'$, which is one of the assumptions. To show $\langle \underline{B} \upharpoonleft \underline{B}' \rangle \bullet' \sqsubseteq S$, we substitute into the second assumption with $\langle \underline{B} \upharpoonleft \underline{B}' \rangle \bullet' \sqsubseteq \bullet'$, which is true by downcast left. □

Casts satisfy an identity and composition law:

Theorem 72 (Casts (de)composition)**.** *For any $A \sqsubseteq A' \sqsubseteq A''$ and $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$:*

1. $x : A \vdash \langle A \leftharpoonup A \rangle x \sqsupseteq\sqsubseteq x : A$
2. $x : A \vdash \langle A'' \leftharpoonup A \rangle x \sqsupseteq\sqsubseteq \langle A'' \leftharpoonup A' \rangle \langle A' \leftharpoonup A \rangle x : A''$
3. $\bullet : \underline{B} \vdash \langle \underline{B} \upharpoonleft \underline{B} \rangle \bullet \sqsupseteq\sqsubseteq \bullet : \underline{B}$
4. $\bullet : \underline{B}'' \vdash \langle \underline{B} \upharpoonleft \underline{B}'' \rangle \bullet \sqsupseteq\sqsubseteq \langle \underline{B} \upharpoonleft \underline{B}' \rangle (\langle \underline{B}' \upharpoonleft \underline{B}'' \rangle \bullet) : \underline{B} \sqsubseteq \underline{B}$

Proof. We use Theorem 71 in all cases, and show that the right-hand side has the universal property of the left.

1. Both parts expand to showing $x \sqsubseteq x : A \sqsubseteq A \vdash x \sqsubseteq x : A \sqsubseteq A$, which is true by assumption.
2. First, we need to show $x \sqsubseteq \langle A'' \leftharpoonup A' \rangle (\langle A' \leftharpoonup A \rangle x) : A \sqsubseteq A''$. By upcast right, it suffices to show $x \sqsubseteq \langle A' \leftharpoonup A \rangle x : A \sqsubseteq A'$, which is also true by upcast right.

   For $x \sqsubseteq x'' : A \sqsubseteq A'' \vdash \langle A'' \leftharpoonup A' \rangle (\langle A' \leftharpoonup A \rangle x) \sqsubseteq x''$, by upcast left twice, it suffices to show $x \sqsubseteq x'' : A \sqsubseteq A''$, which is true by assumption.
3. Both parts expand to showing $\bullet : \underline{B} \vdash \bullet \sqsubseteq \bullet : \underline{B}$, which is true by assumption.
4. To show $\bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}'' \vdash \bullet \sqsubseteq \langle \underline{B} \upharpoonleft \underline{B}' \rangle (\langle \underline{B}' \upharpoonleft \underline{B}'' \rangle \bullet)$, by downcast right (twice), it suffices to show $\bullet : \underline{B} \sqsubseteq \bullet'' : \underline{B}'' \vdash \bullet \sqsubseteq \bullet'' : \underline{B} \sqsubseteq \underline{B}''$, which is true by assumption. Next, we have to show $\langle \underline{B} \upharpoonleft \underline{B}' \rangle (\langle \underline{B}' \upharpoonleft \underline{B}'' \rangle \bullet) \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}''$, and by downcast left, it suffices to show $\langle \underline{B}' \upharpoonleft \underline{B}'' \rangle \bullet \sqsubseteq \bullet : \underline{B}' \sqsubseteq \underline{B}''$, which is also true by downcast left.

□

In particular, this composition property implies that the casts into and out of the dynamic type are coherent, for example if $A \sqsubseteq A'$ then $\langle ? \leftharpoonup A \rangle x \sqsupseteq\sqsubseteq \langle ? \leftharpoonup A' \rangle \langle A' \leftharpoonup A \rangle x$.

Theorem 73 (Casts form Galois Connections)**.** *If $A \sqsubseteq A'$, then the following hold*

1. $\bullet' : \underline{F}A' \vdash \langle\!\langle \underline{F}A' \nwarrow \underline{F}A \rangle\!\rangle \langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet' \sqsubseteq \bullet' : \underline{F}A'$
2. $\bullet : \underline{F}A \vdash \bullet \sqsubseteq \langle \underline{F}A \swarrow \underline{F}A' \rangle \langle\!\langle \underline{F}A' \nwarrow \underline{F}A \rangle\!\rangle \bullet : \underline{F}A$

If $\underline{B} \sqsubseteq \underline{B}'$, then the following hold

1. $x : U\underline{B}' \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle \langle\!\langle U\underline{B} \swarrow U\underline{B}' \rangle\!\rangle x \sqsubseteq x : U\underline{B}'$
2. $x : U\underline{B} \vdash x \sqsubseteq \langle\!\langle U\underline{B} \swarrow U\underline{B}' \rangle\!\rangle \langle U\underline{B}' \nwarrow U\underline{B} \rangle x : U\underline{B}$

Proof.

1. By η for F types, $\bullet' : \underline{F}A' \vdash \bullet' \sqsupseteq\sqsubseteq x' \leftarrow \bullet'; \mathtt{ret}\ x' : \underline{F}A'$, so it suffices to show

   $$x \leftarrow \langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet'; \mathtt{ret}\ (\langle A' \nwarrow A \rangle x) \sqsubseteq x' : A' \leftarrow \bullet'; \mathtt{ret}\ x'$$

   By congruence, it suffices to show $\langle \underline{F}A \swarrow \underline{F}A' \rangle \bullet' \sqsubseteq \bullet' : \underline{F}A \sqsubseteq \underline{F}A'$, which is true by downcast left, and $x \sqsubseteq x' : A \sqsubseteq A' \vdash \mathtt{ret}\ (\langle A' \nwarrow A \rangle x) \sqsubseteq \mathtt{ret}\ x' : A'$, which is true by congruence for ret, upcast left, and the assumption.

2. By η for F types, it suffices to show

$$\bullet : \underline{F}A \vdash \bullet \leftarrow x; \mathtt{ret}\ x \sqsubseteq x \leftarrow \bullet; \langle \underline{F}A \swarrow \underline{F}A' \rangle (\mathtt{ret}\ (\langle A' \nwarrow A \rangle x)) : \underline{F}A$$

   so by congruence,

   $$x : A \vdash \mathtt{ret}\ x \sqsubseteq \langle \underline{F}A \swarrow \underline{F}A' \rangle (\mathtt{ret}\ (\langle A' \nwarrow A \rangle x))$$

   By downcast right, it suffices to show

   $$x : A \vdash \mathtt{ret}\ x \sqsubseteq (\mathtt{ret}\ (\langle A' \nwarrow A \rangle x)) : \underline{F}A \sqsubseteq \underline{F}A'$$

   and by congruence

   $$x : A \vdash x \sqsubseteq ((\langle A' \nwarrow A \rangle x)) : A \sqsubseteq A'$$

   which is true by upcast right.

3. By η for U types, it suffices to show

$$x : U\underline{B}' \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle (\mathtt{thunk}\ (\langle \underline{B} \swarrow \underline{B}' \rangle \mathtt{force}\ x)) \sqsubseteq \mathtt{thunk}\ (\mathtt{force}\ x) : U\underline{B}'$$

   By upcast left, it suffices to show

$$x : U\underline{B}' \vdash (\mathtt{thunk}\ (\langle \underline{B} \swarrow \underline{B}' \rangle \mathtt{force}\ x)) \sqsubseteq \mathtt{thunk}\ (\mathtt{force}\ x) : U\underline{B} \sqsubseteq U\underline{B}'$$

   and by congruence

   $$x : U\underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \mathtt{force}\ x \sqsubseteq \mathtt{force}\ x : \underline{B} \sqsubseteq \underline{B}'$$

   which is true by downcast left.

4. By η for U types, it suffices to show

$$x : U\underline{B} \vdash \mathtt{thunk}\ (\mathtt{force}\ x) \sqsubseteq \mathtt{thunk}\ (\langle B \not\swarrow B' \rangle (\mathtt{force}\ (\langle U\underline{B}' \not\nwarrow U\underline{B} \rangle x))) : U\underline{B}$$

and by congruence

$$x : U\underline{B} \vdash (\mathtt{force}\ x) \sqsubseteq (\langle B \not\swarrow B' \rangle (\mathtt{force}\ (\langle U\underline{B}' \not\nwarrow U\underline{B} \rangle x))) : \underline{B}$$

By downcast right, it suffices to show

$$x : U\underline{B} \vdash (\mathtt{force}\ x) \sqsubseteq (\mathtt{force}\ (\langle U\underline{B}' \not\nwarrow U\underline{B} \rangle x)) : \underline{B} \sqsubseteq \underline{B}'$$

and by congruence

$$x : U\underline{B} \vdash x \sqsubseteq (\langle U\underline{B}' \not\nwarrow U\underline{B} \rangle x) : \underline{B} \sqsubseteq \underline{B}'$$

which is true by upcast right.

□

The retract property says roughly that $x \sqsupseteq\sqsubseteq \langle T' \not\swarrow T \rangle \langle T' \not\nwarrow T \rangle x$ (upcast then downcast does not change the behavior), strengthening the $\sqsubseteq$ of Theorem 73. In Figure 5.6, we asserted the retract axiom for casts with the dynamic type. This and the composition property implies the retraction property for general casts:

Theorem 74 (Retract Property for General Casts). *If $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$, then*

1. $\bullet : \underline{F}A \vdash \langle\!\langle \underline{F}A' \not\nwarrow \underline{F}A \rangle\!\rangle \langle \underline{F}A \not\swarrow \underline{F}A' \rangle \bullet \sqsupseteq\sqsubseteq \bullet : \underline{F}A$
2. $x : U\underline{B} \vdash \langle\!\langle U\underline{B} \not\swarrow U\underline{B}' \rangle\!\rangle \langle U\underline{B}' \not\nwarrow U\underline{B} \rangle x \sqsupseteq\sqsubseteq x : U\underline{B}$

Proof. We need only to show the $\sqsubseteq$ direction, because the converse is Theorem 73.

1. Substituting $\mathtt{ret}\ (\langle A' \not\nwarrow A \rangle x)$ into Theorem 73's

$$\bullet : \underline{F}A \vdash \bullet \sqsubseteq x \leftarrow \bullet; \langle \underline{F}A \not\swarrow \underline{F}A' \rangle (\mathtt{ret}\ (\langle A' \not\nwarrow A \rangle x)) : \underline{F}A$$

and β-reducing gives

$$x : A \vdash \mathtt{ret}\ (\langle A' \not\nwarrow A \rangle x) \sqsubseteq \langle \underline{F}A \not\swarrow \underline{F}? \rangle (\mathtt{ret}\ (\langle ? \not\nwarrow A' \rangle \langle A' \not\nwarrow A \rangle x))$$

Using this, after η-expanding $\bullet : \underline{F}A$ on the right and using congruence for bind, it suffices to derive as follows:

$$\begin{array}{lll}
\langle \underline{F}A \not\swarrow \underline{F}A' \rangle (\mathtt{ret}\ (\langle A' \not\nwarrow A \rangle x)) & \sqsubseteq & \text{congruence} \\
\langle \underline{F}A \not\swarrow \underline{F}A' \rangle \langle \underline{F}A' \not\swarrow \underline{F}? \rangle (\mathtt{ret}\ (\langle ? \not\nwarrow A' \rangle \langle A' \not\nwarrow A \rangle x)) & \sqsubseteq & \text{composition} \\
\langle \underline{F}A \not\swarrow \underline{F}? \rangle (\mathtt{ret}\ (\langle ? \not\nwarrow A \rangle x)) & \sqsubseteq & \text{retract axiom for } \langle ? \not\nwarrow A \rangle \\
\mathtt{ret}\ x &&
\end{array}$$

2. After using η for U and congruence, it suffices to show

$$x : U\underline{B} \vdash \langle \underline{B} \swarrow \underline{B}' \rangle(\texttt{force}\ (\langle U\underline{B}' \nwarrow U\underline{B} \rangle x)) \sqsubseteq \texttt{force}\ x : \underline{B}$$

Substituting $x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle x : U\underline{B}'$ into Theorem 73's

$$x : U\underline{B}' \vdash x \sqsubseteq \texttt{thunk}\ (\langle B' \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B}' \rangle x))) : U\underline{B}'$$

gives

$$x : U\underline{B} \vdash \langle U\underline{B}' \nwarrow U\underline{B} \rangle x \sqsubseteq \texttt{thunk}\ (\langle B' \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B}' \rangle \langle U\underline{B}' \nwarrow U\underline{B} \rangle x))) : U\underline{B}'$$

So we have

$$\begin{array}{lll}
\langle B \swarrow B' \rangle(\texttt{force}\ \langle U\underline{B}' \nwarrow U\underline{B} \rangle x) & \sqsubseteq & \\
\langle B \swarrow B' \rangle \texttt{force}\ (\texttt{thunk}\ (\langle B' \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B}' \rangle \langle U\underline{B}' \nwarrow U\underline{B} \rangle x)))) & \sqsubseteq & \beta \\
\langle B \swarrow B' \rangle(\langle B' \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B}' \rangle \langle U\underline{B}' \nwarrow U\underline{B} \rangle x))) & \sqsubseteq & \text{composition} \\
\langle B \swarrow \underline{¿} \rangle(\texttt{force}\ (\langle U\underline{¿} \nwarrow U\underline{B} \rangle x)) & \sqsubseteq & \text{retract axiom for } \langle \underline{B} \swarrow \underline{¿} \rangle \\
\texttt{ret}\ x & \sqsubseteq & \text{composition}
\end{array}$$

□

5.3.3 *Deriving Behavior of Casts*

We now come to the central technical consequence of the axioms of GTT, that we can *derive* the behavior of most casts from just η principles and our definition of upcasts and downcasts as least upper bounds and greatest lower bounds, respectively.

Together, the universal property for casts and the η principles for each type imply that the casts must behave as in "wrapping" cast semantics, which we will demonstrate more explicitly in Chapter 6:

Theorem 75 (Cast Unique Implementation Theorem for $+, \times, \to, \&$)**.** *All of the equivalences in Figure 5.7 are derivable.*

Proof. The proofs are at the end of this subsection, using the upcast/-downcast lemmas 79,80 which we define shortly. □

For each value type connective, we derive the semantics of the upcast and the semantics of the corresponding downcast where $\underline{F}$ is applied to the connective. Dually for the computation type connectives we derive the downcast and the upcast where a U is applied. Note that all of the definitions of casts are essentially the same as the definitions of the operational behavior given in the "wrapping" semantics of gradual typing.

Notably, for the eager product $\times$ and the function type $\to$, we derive that two a priori different implementations both satisfy the specification and so are equivalent. The two implementations differ in that one evaluates the downcast for the left side of the pair first,

$$\langle A_1' + A_2' \leftharpoondown A_1 + A_2 \rangle s \sqsupseteq\sqsubseteq \mathtt{case}\ s\{x_1.\mathtt{inl}\ (\langle A_1' \leftharpoondown A_1\rangle x_1) \mid x_2.\mathtt{inr}\ (\langle A_2' \leftharpoondown A_2\rangle x_2)\}$$

$$\begin{array}{rcl}\langle \underline{F}(A_1' + A_2') \swarrow \underline{F}(A_1 + A_2)\rangle \bullet & \sqsupseteq\sqsubseteq & (s : (A_1' + A_2')) \leftarrow \bullet; \mathtt{case}\ s \\ & & \{x_1'.x_1 \leftarrow (\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle(\mathtt{ret}\ x_1')); \\ & & \quad \mathtt{ret}\ (\mathtt{inl}\ x_1) \\ & & \mid x_2'.x_2 \leftarrow (\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle(\mathtt{ret}\ x_2'));\} \\ & & \quad \mathtt{ret}\ (\mathtt{inr}\ x_2)\end{array}$$

$$\langle A_1' \times A_2' \leftharpoondown A_1 \times A_2\rangle p \sqsupseteq\sqsubseteq \mathtt{let}\ (x_1, x_2) = p; (\langle A_1' \leftharpoondown A_1\rangle x_1, \langle A_2' \leftharpoondown A_2\rangle x_2)$$

$$\begin{array}{rcl}\langle \underline{F}(A_1' \times A_2') \swarrow \underline{F}(A_1 \times A_2)\rangle \bullet & \sqsupseteq\sqsubseteq & p' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = p'; \\ & & x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \mathtt{ret}\ x_1'; \\ & & x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \mathtt{ret}\ x_2'; \mathtt{ret}\ (x_1, x_2) \\ \\ & \sqsupseteq\sqsubseteq & p' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = p'; \\ & & x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle \mathtt{ret}\ x_2'; \\ & & x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle \mathtt{ret}\ x_1'; \mathtt{ret}\ (x_1, x_2)\end{array}$$

$$\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle \bullet \sqsupseteq\sqsubseteq (\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle \pi \bullet, \langle \underline{B}_2 \swarrow \underline{B}_2'\rangle \pi' \bullet)$$

$$\begin{array}{l}\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoondown U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle p \\ \quad \sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ (\langle U\underline{B}_1' \leftharpoondown U\underline{B}_1\rangle(\mathtt{thunk}\ \pi(\mathtt{force}\ p))) \quad \} \\ \qquad\qquad \mid \pi' \mapsto \mathtt{force}\ (\langle U\underline{B}_2' \leftharpoondown U\underline{B}_2\rangle(\mathtt{thunk}\ \pi'(\mathtt{force}\ p)))\end{array}$$

$$\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle \bullet \sqsupseteq\sqsubseteq \lambda x.\langle \underline{B} \swarrow \underline{B}'\rangle(\bullet\,(\langle A' \leftharpoondown A\rangle x))$$

$$\begin{array}{l}\langle U(A' \to \underline{B}') \leftharpoondown U(A \to \underline{B})\rangle f \sqsupseteq\sqsubseteq \mathtt{thunk}\ (\lambda x'.x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle(\mathtt{ret}\ x'); \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{force}\ (\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle(\mathtt{thunk}\ ((\mathtt{force}\ f)\, x))) \\ \\ \quad \sqsupseteq\sqsubseteq \mathtt{thunk}\ (\lambda x'.\mathtt{force}\ \langle U\underline{B}' \leftharpoondown U\underline{B}\rangle(\mathtt{thunk}\ (x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle(\mathtt{ret}\ x'); \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathtt{force}\ f)\, x)))\end{array}$$

Figure 5.7: Derivable Cast Behavior for $+, \times, \&, \to$

whereas the other evaluates the downcast for the right side of the pair first. It makes sense operationally that these two are equivalent, since all either can do is error. If we were to incorporate blame, then each side might raise a different error but would blame the same party. There is a similar (non-)choice for the function type, which is intuitively the choice between enforcing domain or codomain first. This might seem unusual at first glance, since how can the codomain contract be enforced before the function is called with an input? The answer is that when the $\underline{B} = \underline{F}A$ then the upcast of the output will first force the evaluation of the function, but when $\underline{B}$ is a CBPV function type we see a similar ambiguity to the strict product. In particular, a call-by-value two-argument function $A_1, A_2 \to A_3$ can be represented by the CBPV type $A_1 \to A_2 \to \underline{F}A_3$, and this ambiguity between enforcing the domain and codomain first arises as the ambiguity of in which order a multi-argument function contract should enforce its multiple domain contracts. As with the product, the orderings turn out to be equivalent.

We can similarly derive cast implementations for the "double shifts":

Theorem 76 (Cast Unique Implementation Theorem for $U\underline{F}, \underline{F}U$)**.** *Let $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$.*

1. $x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A\rangle x \sqsupseteq\sqsubseteq \mathtt{thunk}\ (\langle\!\langle \underline{F}A' \nwarrow \underline{F}A\rangle\!\rangle(\mathtt{force}\ x)) : U\underline{F}A'$
2. $\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle \bullet \sqsupseteq\sqsubseteq x' : U\underline{B}' \leftarrow \bullet; \mathtt{ret}\ (\langle U\underline{B} \swarrow U\underline{B}'\rangle x)$

Proof. Again, at the end of this subsection. □

While we can prove each of these cases directly, the proofs are fairly repetitive and similar. Instead we package up the proof principle into a couple of lemmas which abstract over the details of the proof. First, since all of these proof principles are parameterized, we need to formally define parameterized types in order to prove our general lemmas.

Definition 77. Let a *type constructor* C be a (value or computation) type that well-formed according to the grammar in Figure 5.1 with additional hypotheses X val type and $\underline{Y}$ comp type standing for value or computation types, respectively. We write $C[A/X]$ and $C[\underline{B}/\underline{Y}]$ for the substitution of a type for a variable.

For example,

$$X_1 \text{ val type}, X_2 \text{ val type} \vdash X_1 + X_2 \text{ val type}$$
$$\underline{Y} \text{ comp type} \vdash U\underline{Y} \text{ val type}$$
$$X_1 \text{ val type}, X_2 \text{ val type} \vdash \underline{F}(X_1 + X_2) \text{ comp type}$$

are type constructors.

Observe that all type constructors are monotone in type precision, because we included a congruence rule for every type constructor in Figure 5.3:

Lemma 78 (Monotonicity of Type Constructors)**.** *For any type constructor X val type $\vdash C$, if $A \sqsubseteq A'$ then $C[A/X] \sqsubseteq C[A'/x]$. For any type constructor $\underline{Y}$ comp type $\vdash C$, if $\underline{B} \sqsubseteq \underline{B}'$ then $C[\underline{B}/\underline{Y}] \sqsubseteq C[\underline{B}'/\underline{Y}]$.*

Proof. Induction on C. In the case for a variable X or $\underline{Y}$, $A \sqsubseteq A'$ or $\underline{B} \sqsubseteq \underline{B}'$ by assumption. In all other cases, the result follows from the inductive hypotheses and the congruence rule for type precision for the type constructor (Figure 5.3). For example, in the case for $+$, $A_1[A/x] \sqsubseteq A_1[A'/x]$ and $A_2[A/x] \sqsubseteq A_2[A'/x]$, so $A_1[A/x] + A_2[A/x] \sqsubseteq A_1[A'/x] + A_2[A'/x]$. □

The following lemma helps show that a complex value

$$\langle\!\langle C[A_i'/X_i, \underline{B}_i'/\underline{Y}_i] \leftharpoondown C[A_i/X_i, \underline{B}_i/\underline{Y}_i] \rangle\!\rangle$$

is an upcast from $C[A_i/X_i, \underline{B}_i/\underline{Y}_i]$ to $C[A_i'/X_i, \underline{B}_i'/\underline{Y}_i]$. It reduces to verification of 3 properties: well-typedness, monotonicity and identity extension. Of these, only identity extension is non-trivial to prove.

Lemma 79 (Upcast Lemma)**.** *Let X_1 val type, $\dots$ X_n val type, $\underline{Y}_1$ comp type, $\dots$ $\underline{Y}_n$ comp type $\vdash C$ val type be a value type constructor. We abbreviate the instantiation $C[A_1/X_1, \dots, A_n/X_n, \underline{B}_1/\underline{Y}_i, \dots, \underline{B}_m/\underline{Y}_m]$ by $C[\overline{A_i}, \overline{\underline{B}_i}]$.*

Suppose $\langle\!\langle C[\overline{A_i'}, \overline{\underline{B}_i'}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle -$ is a complex value (depending on C and each of $\overline{A_i}$, $\overline{A_i'}$, $\overline{\underline{B}_i}$,$\overline{\underline{B}_i'}$) such that

1. *(Well-typedness) For all value types $\overline{A_i}$ and $\overline{A_i'}$ with each of $A_i \sqsubseteq A_i'$, and all computation types $\overline{\underline{B}_i}$ and $\overline{\underline{B}_i'}$ with all $\underline{B}_i \sqsubseteq \underline{B}_i'$,*

$$x : C[\overline{A_i}, \overline{\underline{B}_i}] \vdash \langle\!\langle C[\overline{A_i'}, \overline{\underline{B}_i'}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x : C[\overline{A_i'}, \overline{\underline{B}_i'}]$$

2. *(Monotonicity) For all $\overline{A_i^l}$,$\overline{A_i^r}$,$\overline{A_i^{l\prime}}$,$\overline{A_i^{r\prime}}$,$\overline{\underline{B}_i^{\,l}}$,$\overline{\underline{B}_i^{\,r}}$,$\overline{\underline{B}_i^{\,l\prime}}$,$\overline{\underline{B}_i^{\,r\prime}}$, such that each of $A_i^l \sqsubseteq A_i^r$, $A_i^l \sqsubseteq A_i^{l\prime}$, $A_i^r \sqsubseteq A_i^{r\prime}$ and similarly $\underline{B}_i^l \sqsubseteq \underline{B}_i^r$, $\underline{B}_i^l \sqsubseteq \underline{B}_i^{l\prime}$, $\underline{B}_i^r \sqsubseteq \underline{B}_i^{r\prime}$,*

$$\Phi \vdash \langle\!\langle C[\overline{A_i^r}, \overline{\underline{B}_i^r}] \leftharpoondown C[\overline{A_i^l}, \overline{\underline{B}_i^l}] \rangle\!\rangle x \sqsubseteq \langle\!\langle C[\overline{A_i^{r\prime}}, \overline{\underline{B}_i^{r\prime}}] \leftharpoondown C[\overline{A_i^{l\prime}}, \overline{\underline{B}_i^{l\prime}}] \rangle\!\rangle x' : C[\overline{A_i^r}, \overline{\underline{B}_i^r}] \sqsubseteq C[\overline{A_i^{r\prime}}, \overline{\underline{B}_i^{r\prime}}]$$

   where $\Phi = x \sqsubseteq x' : C[\overline{A_i^l}, \overline{\underline{B}_i^l}] \sqsubseteq C[\overline{A_i^r}, \overline{\underline{B}_i^r}]$

3. *(Identity Extension) For all value types $\overline{A_1}$ and all computation types $\overline{\underline{B}_i}$,*

$$x : C[\overline{A_i}, \overline{\underline{B}_i}] \vdash \langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsupseteq\sqsubseteq x : C[\overline{A_i}, \overline{\underline{B}_i}]$$

Then $\langle\!\langle C[\overline{A_i'}, \overline{\underline{B}_i'}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle$ satisfies the universal property of an upcast, so by Theorem 71

$$x : C[\overline{A_i}, \overline{\underline{B}_i}] \vdash \langle\!\langle C[\overline{A_i'}, \overline{\underline{B}_i'}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsupseteq\sqsubseteq \langle C[\overline{A_i'}, \overline{\underline{B}_i'}] \leftharpoondown C[\overline{A_i}, \overline{\underline{B}_i}] \rangle x : C[\overline{A_i'}, \overline{\underline{B}_i'}]$$

Moreover, the left-to-right direction uses only the left-to-right direction of identity extension, and the right-to-left uses only the right-to-left direction.

Proof. First, we show that $\langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle$ satisfies the universal property of an upcast.

To show

$$x \sqsubseteq x' : C[\overline{A_i}, \overline{\underline{B}_i}] \sqsubseteq C[\overline{A'_i}, \overline{\underline{B}'_i}] \vdash \langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsubseteq x' : C[\overline{A'_i}, \overline{\underline{B}'_i}]$$

monotonicity gives that

$$\langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsubseteq \langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle\!\rangle x' : C[\overline{A'_i}, \overline{\underline{B}'_i}]$$

but by the left-to-right direction of identity extension the right hand side is more precise than x', so transitivity gives the result.

$$\langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsubseteq x'$$

To show

$$x \sqsubseteq \langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x : C[\overline{A_i}, \overline{\underline{B}_i}] \sqsubseteq C[\overline{A'_i}, \overline{\underline{B}'_i}]$$

By monotonicity, we have

$$\langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x \sqsubseteq \langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x : C[\overline{A_i}, \overline{\underline{B}_i}] \sqsubseteq C[\overline{A'_i}, \overline{\underline{B}'_i}]$$

so transitivity with the right-to-left direction of identity extension gives the result:

$$x \sqsubseteq \langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle x$$

Then Theorem 71 implies that $\langle\!\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle$ is equivalent to $\langle C[\overline{A'_i}, \overline{\underline{B}'_i}] \nwarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle$. □

We have then also the exact dual lemma for downcasts:

Lemma 80 (Downcast Lemma)**.** *Let X_1 val type, $\dots$ X_n val type, $\underline{Y}_1$ comp type, $\dots$ $\underline{Y}_n$ comp type $\vdash$ C comp type be a computation type constructor. We abbreviate the instantiation*
$C[A_1/X_1, \dots, A_n/X_n, \underline{B}_1/\underline{Y}_i, \dots, \underline{B}_m/\underline{Y}_m]$ by $C[\overline{A_i}, \overline{\underline{B}_i}]$.

Suppose $\langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \swarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle\!\rangle -$ is a complex stack (depending on C and each of $\overline{A_i}$, $\overline{A'_i}$, $\overline{\underline{B}_i}$,$\overline{\underline{B}'_i}$) such that

1. *(Well-typedness) For all value types $\overline{A_i}$ and $\overline{A'_i}$ with each of $A_i \sqsubseteq A'_i$, and all computation types $\overline{\underline{B}_i}$ and $\overline{\underline{B}'_i}$ with all $\underline{B}_i \sqsubseteq \underline{B}'_i$,*

$$\bullet : C[\overline{A'_i}, \overline{\underline{B}'_i}] \vdash \langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \swarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle\!\rangle \bullet : C[\overline{A_i}, \overline{\underline{B}_i}]$$

2. *(Monotonicity) For all $\overline{A^l_i}$,$\overline{A^r_i}$,$\overline{A^{l\prime}_i}$,$\overline{A^{r\prime}_i}$,$\overline{\underline{B}^l_i}$,$\overline{\underline{B}^r_i}$,$\overline{\underline{B}^{l\prime}_i}$,$\overline{\underline{B}^{r\prime}_i}$, such that each of $A^l_i \sqsubseteq A^r_i$, $A^l_i \sqsubseteq A^{l\prime}_i$, $A^r_i \sqsubseteq A^{r\prime}_i$ and similarly $\underline{B}^l_i \sqsubseteq \underline{B}^r_i$, $\underline{B}^l_i \sqsubseteq \underline{B}^{l\prime}_i$, $\underline{B}^r_i \sqsubseteq \underline{B}^{r\prime}_i$,*

$$\Phi \vdash \langle\!\langle C[\overline{A^l_i}, \overline{\underline{B}^l_i}] \swarrow C[\overline{A^r_i}, \overline{\underline{B}^r_i}] \rangle\!\rangle \bullet \sqsubseteq \langle\!\langle C[\overline{A^{l\prime}_i}, \overline{\underline{B}^{l\prime}_i}] \swarrow C[\overline{A^{r\prime}_i}, \overline{\underline{B}^{r\prime}_i}] \rangle\!\rangle \bullet' : C[\overline{A^l_i}, \overline{\underline{B}^l_i}] \sqsubseteq C[\overline{A^r_i}, \overline{\underline{B}^r_i}]$$

where $\Phi = \bullet \sqsubseteq \bullet' : C[\overline{A^r_i}, \overline{\underline{B}^r_i}] \sqsubseteq C[\overline{A^{r\prime}_i}, \overline{\underline{B}^{r\prime}_i}]$

3. *(Identity Extension) For all value types $\overline{A_1}$ and all computation types $\overline{\underline{B}_i}$,*

$$\bullet : C[\overline{A_i}, \overline{\underline{B}_i}] \vdash \langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \not\swarrow C[\overline{A_i}, \overline{\underline{B}_i}] \rangle\!\rangle \bullet \sqsupseteq\sqsubseteq \bullet : C[\overline{A_i}, \overline{\underline{B}_i}]$$

Then $\langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \not\swarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle\!\rangle$ satisfies the universal property of a downcast, so by Theorem 71

$$\bullet : C[\overline{A'_i}, \overline{\underline{B}'_i}] \vdash \langle\!\langle C[\overline{A_i}, \overline{\underline{B}_i}] \not\swarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle\!\rangle \bullet \sqsupseteq\sqsubseteq \langle C[\overline{A_i}, \overline{\underline{B}_i}] \not\swarrow C[\overline{A'_i}, \overline{\underline{B}'_i}] \rangle \bullet : C[\overline{A_i}, \overline{\underline{B}_i}]$$

Moreover, the left-to-right direction uses only the left-to-right direction of identity extension, and the right-to-left uses only the right-to-left direction of identity extension.

Proof. The proof is the exact dual of the proof of Lemma 79. □

As an example derivation we prove the case for a downcast for function types:

$$\langle A \to \underline{B} \not\swarrow A' \to \underline{B}' \rangle \bullet \sqsupseteq\sqsubseteq \lambda x. \langle \underline{B} \not\swarrow \underline{B}' \rangle (\bullet\, (\langle A' \nwarrow A \rangle x))$$

Here the type constructor is X val type, $\underline{Y}$ comp type $\vdash X \to Y$ comp type. We apply the downcast lemma with the definition being

$$\langle\!\langle A \to \underline{B} \not\swarrow A' \to \underline{B}' \rangle\!\rangle = \lambda x. \langle \underline{B} \not\swarrow \underline{B}' \rangle (\bullet\, (\langle A' \nwarrow A \rangle x))$$

Then well-typedness clearly holds, and monotonicity follows by congruence for all constructors and Lemma 68 for the casts. Finally, for identity extension we need to show

$$\lambda x. \langle \underline{B} \not\swarrow \underline{B} \rangle (\bullet\, (\langle A \nwarrow A \rangle x)) \sqsupseteq\sqsubseteq \bullet : A \to \underline{B}$$

First, by the decomposition theorem 72 this is equivalent to

$$\lambda x. \bullet\ x \sqsupseteq\sqsubseteq \bullet : A \to \underline{B}$$

Which is precisely η equivalence for $\to$. The cases for the other connectives proceed similarly.

5.3.4 *Proof of Theorem 75*

Proof. 1. Sums upcast. We use Lemma 79 with the type constructor X_1 val type, X_2 val type $\vdash X_1 + X_2$ val type. Suppose $A_1 \sqsubseteq A'_1$ and $A_2 \sqsubseteq A'_2$ and let

$$s : A_1 + A_2 \vdash \langle\!\langle A'_1 + A'_2 \nwarrow A_1 + A_2 \rangle\!\rangle s : A'_1 + A'_2$$

stand for

$$\mathtt{case}\ s \{ x_1.\mathtt{inl}\ (\langle A'_1 \nwarrow A_1 \rangle x_1) \mid x_2.\mathtt{inr}\ (\langle A'_2 \nwarrow A_2 \rangle x_2) \}$$

This clearly satisfies the typing requirement and monotonicity.

Finally, for identity extension, we need to show

$$\texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1 \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2 \nwarrow A_2\rangle x_2)\} \sqsupseteq\sqsubseteq s$$

which is true because $\langle A_1 \nwarrow A_1\rangle$ and $\langle A_2 \nwarrow A_2\rangle$ are the identity, and using "weak η" for sums, $\texttt{case}\ s\{x_1.\texttt{inl}\ x_1 \mid x_2.\texttt{inr}\ x_2\} \sqsupseteq\sqsubseteq x$, which is the special case of the η rule in Figure 5.6 for the identity complex value:

$$\begin{array}{rl} \texttt{case}\ s\{x_1.\texttt{inl}\ (\langle A_1 \nwarrow A_1\rangle x_1) \mid x_2.\texttt{inr}\ (\langle A_2 \nwarrow A_2\rangle x_2)\} & \sqsupseteq\sqsubseteq \\ \texttt{case}\ s\{x_1.\texttt{inl}\ (x_1) \mid x_2.\texttt{inr}\ (x_2)\} & \sqsupseteq\sqsubseteq \\ s & \end{array}$$

2. Sums downcast. We use the downcast lemma with X_1 val type, X_2 val type $\vdash \underline{F}(X_1 + X_2)$ comp type. Let

$$\bullet' : \underline{F}(A_1' + A_2') \vdash \langle\!\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2')\rangle\!\rangle \bullet' : \underline{F}(A_1 + \underline{A}_2)$$

stand for

$$(s : (A_1' + A_2')) \leftarrow \bullet; \texttt{case}\ s\{x_1'.x_1 \leftarrow (\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle(\texttt{ret}\ x_1')); \texttt{ret}\ (\texttt{inl}\ x_1) \mid \ldots\}$$

(where, as in the theorem statement, inr branch is analogous).
This clearly satisfies typing and monotonicity.

Finally, for identity extension, we show

$$\begin{array}{ll} (s : (A_1 + A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.x_1 \leftarrow (\langle \underline{F}A_1 \swarrow \underline{F}A_1\rangle(\texttt{ret}\ x_1)); \texttt{ret}\ (\texttt{inl}\ x_1) \mid \ldots\} & \sqsupseteq\sqsubseteq \\ (s : (A_1 + A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.x_1 \leftarrow ((\texttt{ret}\ x_1)); \texttt{ret}\ (\texttt{inl}\ x_1) \mid \ldots\} & \sqsupseteq\sqsubseteq \\ (s : (A_1 + A_2)) \leftarrow \bullet; \texttt{case}\ s\{x_1.\texttt{ret}\ (\texttt{inl}\ x_1) \mid x_2.\texttt{ret}\ (\texttt{inr}\ x_2)\} & \sqsupseteq\sqsubseteq \\ (s : (A_1 + A_2)) \leftarrow \bullet; \texttt{ret}\ s & \sqsupseteq\sqsubseteq \\ \bullet & \end{array}$$

using the downcast identity, β for $\underline{F}$ types, η for sums, and η for $\underline{F}$ types.

3. Eager product upcast. We use Lemma 79 with the type constructor X_1 val type, X_2 val type $\vdash X_1 \times X_2$ val type. Let

$$p : A_1 \times A_2 \vdash \langle\!\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle\!\rangle s : A_1' \times A_2'$$

stand for

$$\texttt{let}\ (x_1, x_2) = p; (\langle A_1' \nwarrow A_1\rangle x_1, \langle A_2' \nwarrow A_2\rangle x_2)$$

which clearly satisfies the typing requirement and monotonicity.

Finally, for identity extension, using η for products and the fact that $\langle A \nwarrow A\rangle$ is the identity, we have

$$\texttt{let}\ (x_1, x_2) = p; (\langle A_1 \nwarrow A_1\rangle x_1, \langle A_2 \nwarrow A_2\rangle x_2) \sqsupseteq\sqsubseteq \texttt{let}\ (x_1, x_2) = p; (x_1, x_2) \sqsupseteq\sqsubseteq p$$

4. Eager product downcast.

   We use the downcast lemma with X_1 val type, X_2 val type $\vdash$ $\underline{F}(X_1 \times X_2)$ comp type. Let

   $$\bullet' : \underline{F}(A_1' \times A_2') \vdash \langle\!\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2') \rangle\!\rangle \bullet' : \underline{F}(A_1 \times \underline{A}_2)$$

   stand for

$$p' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = p'; x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle \mathtt{ret}\ x_1'; x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle \mathtt{ret}\ x_2'; \mathtt{ret}\ (x_1, x_2)$$

   which clearly satisfies the typing requirement and monotonicity.

   Finally, for identity extension, we show

$$\begin{array}{ll}
p \leftarrow \bullet; \mathtt{let}\ (x_1, x_2) = p; x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1 \rangle \mathtt{ret}\ x_1; x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle \mathtt{ret}\ x_2; \mathtt{ret}\ (x_1, x_2) & \sqsupseteq\sqsubseteq \\
p \leftarrow \bullet; \mathtt{let}\ (x_1, x_2) = p; x_1 \leftarrow \mathtt{ret}\ x_1; x_2 \leftarrow \mathtt{ret}\ x_2; \mathtt{ret}\ (x_1, x_2) & \sqsupseteq\sqsubseteq \\
p \leftarrow \bullet; \mathtt{let}\ (x_1, x_2) = p; \mathtt{ret}\ (x_1, x_2) & \sqsupseteq\sqsubseteq \\
p \leftarrow \bullet; \mathtt{ret}\ p & \sqsupseteq\sqsubseteq \\
\bullet &
\end{array}$$

   using the downcast identity, β for $\underline{F}$ types, η for eager products, and η for $\underline{F}$ types.

   An analogous argument works if we sequence the downcasts of the components in the opposite order:

$$p' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = p'; x_2 \leftarrow \langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle \mathtt{ret}\ x_2'; x_1 \leftarrow \langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle \mathtt{ret}\ x_1'; \mathtt{ret}\ (x_1, x_2)$$

   (the only facts about downcasts used above are congruence and the downcast identity), which shows that these two implementations of the downcast are themselves equiprecise.

5. Lazy product downcast. We use Lemma 80 with the type constructor $\underline{Y}_1$ comp type, $\underline{Y}_2$ comp type $\vdash$ $\underline{Y}_1 \,\&\, \underline{Y}_2$ val type. Let

   $$\bullet' : \underline{B}_1' \,\&\, \underline{B}_2' \vdash \langle\!\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1 \,\&\, \underline{B}_2 \rangle\!\rangle \bullet' : \underline{B}_1 \,\&\, \underline{B}_2$$

   stand for

   $$(\langle \underline{B}_1 \swarrow \underline{B}_1' \rangle \pi \bullet', \langle \underline{B}_2 \swarrow \underline{B}_2' \rangle \pi' \bullet')$$

   which clearly satisfies the typing requirement and monotonicity.

   For identity extension, we have, using $\langle \underline{B} \swarrow \underline{B} \rangle$ is the identity and η for &,

   $$(\langle \underline{B}_1 \swarrow \underline{B}_1 \rangle \pi \bullet, \langle \underline{B}_2 \swarrow \underline{B}_2 \rangle \pi' \bullet) \sqsupseteq\sqsubseteq (\pi \bullet, \pi' \bullet) \sqsupseteq\sqsubseteq \bullet$$

6. Lazy product upcast.

   We use Lemma 79 with the type constructor $\underline{Y}_1$ comp type, $\underline{Y}_2$ comp type $\vdash$ $U(\underline{Y}_1 \,\&\, \underline{Y}_2)$ val type. Let

   $$p : U(\underline{B}_1 \,\&\, \underline{B}_2) \vdash \langle\!\langle U(\underline{B}_1 \,\&\, \underline{B}_2) \nwarrow U(\underline{B}_1 \,\&\, \underline{B}_2) \rangle\!\rangle p : U(\underline{B}_1' \,\&\, \underline{B}_2')$$

stand for

$$\texttt{thunk}\ (\texttt{force}\ (\langle U\underline{B}_1' \nwarrow U\underline{B}_1\rangle(\texttt{thunk}\ \pi(\texttt{force}\ p))), \texttt{force}\ (\langle U\underline{B}_2' \nwarrow U\underline{B}_2\rangle(\texttt{thunk}\ \pi'(\texttt{force}\ p))))$$

which clearly satisfies the typing requirement and monotonicity.

Finally, for identity extension, using η for *times*, β and η for U types, and the fact that $\langle A \nwarrow A\rangle$ is the identity, we have

$$\begin{aligned}
\texttt{thunk}\ (\texttt{force}\ (\langle U\underline{B}_1 \nwarrow U\underline{B}_1\rangle(\texttt{thunk}\ \pi(\texttt{force}\ p))), \texttt{force}\ (\langle U\underline{B}_2 \nwarrow U\underline{B}_2\rangle(\texttt{thunk}\ \pi'(\texttt{force}\ p)))) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\texttt{force}\ (\texttt{thunk}\ \pi(\texttt{force}\ p)), \texttt{force}\ (\texttt{thunk}\ \pi'(\texttt{force}\ p))) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\pi(\texttt{force}\ p), \pi'(\texttt{force}\ p)) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\texttt{force}\ p) &\sqsupseteq\sqsubseteq\\
p &
\end{aligned}$$

7. Function downcast.

   We use Lemma 80 with the type constructor X val type, $\underline{Y}$ comp type $\vdash$ $X \to \underline{Y}$ comp type. Let

   $$\bullet' : A' \to \underline{B}' \vdash \langle\!\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle\!\rangle \bullet' : A \to \underline{B}$$

   stand for

   $$\lambda x.\langle \underline{B} \swarrow \underline{B}'\rangle(\bullet\,(\langle A' \nwarrow A\rangle x))$$

   which clearly satisfies the typing requirement and monotonicity.

   For identity extension, we have, using $\langle A \nwarrow A\rangle$ and $\langle \underline{B} \swarrow \underline{B}\rangle$ are the identity and η for $\to$,

   $$\lambda x.\langle \underline{B} \swarrow \underline{B}\rangle(\bullet\,(\langle A \nwarrow A\rangle x)) \quad \sqsupseteq\sqsubseteq \lambda x.(\bullet\,(x)) \quad \sqsupseteq\sqsubseteq \bullet$$

8. Function upcast. We use Lemma 79 with the type constructor $\underline{X}$ val type, $\underline{Y}$ comp type $\vdash$ $U(\underline{X} \to \underline{Y})$ val type. Suppose $A \sqsubseteq A'$ as value types and $\underline{B} \sqsubseteq \underline{B}'$ as computation types and let

   $$p : U(A \to \underline{B}) \vdash \langle\!\langle U(A \to \underline{B}) \nwarrow U(A \to \underline{B})\rangle\!\rangle p : U(A' \to \underline{B}')$$

   stand for

$$\texttt{thunk}\ (\lambda x'.x \leftarrow \langle \underline{F}A \swarrow \underline{F}A'\rangle(\texttt{ret}\ x'); \texttt{force}\ (\langle U\underline{B}' \nwarrow U\underline{B}\rangle(\texttt{thunk}\ (\texttt{force}\ (f)\,x))))$$

   which clearly satisfies the typing requirement and monotonicity.

   Finally, for identity extension, using η for $\to$, β for F types and β/η for U types, and the fact that $\langle \underline{B} \nwarrow \underline{B}\rangle$ and $\langle A \swarrow A\rangle$ are the identity, we have

$$\begin{aligned}
\texttt{thunk}\ (\lambda x.x \leftarrow \langle \underline{F}A \swarrow \underline{F}A\rangle(\texttt{ret}\ x); \texttt{force}\ (\langle U\underline{B} \nwarrow U\underline{B}\rangle(\texttt{thunk}\ (\texttt{force}\ (f)\,x)))) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\lambda x.x \leftarrow (\texttt{ret}\ x); \texttt{force}\ (\texttt{thunk}\ (\texttt{force}\ (f)\,x))) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\lambda x.\texttt{force}\ (\texttt{thunk}\ (\texttt{force}\ (f)\,x))) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\lambda x.(\texttt{force}\ (f)\,x)) &\sqsupseteq\sqsubseteq\\
\texttt{thunk}\ (\texttt{force}\ (f)) &\sqsupseteq\sqsubseteq\\
f &
\end{aligned}$$

9. $z : 0 \vdash \langle A \nwarrow 0\rangle z \sqsupseteq\sqsubseteq \mathtt{absurd}\ z : A$ is immediate by η for 0 on the map $z : 0 \vdash \langle A \nwarrow 0\rangle z : A$.

□

PROOF OF THEOREM 76

Proof. 1. We apply the upcast lemma with the type constructor X val type $\vdash U\underline{F}X$ val type. The term $\mathtt{thunk}\ (\langle\!\langle \underline{F}A' \nwarrow \underline{F}A\rangle\!\rangle(\mathtt{force}\ x))$ has the correct type and clearly satisfies monotonicity. Finally, for identity extension, we have

$$\begin{array}{rl} \mathtt{thunk}\ (\langle\!\langle \underline{F}A \nwarrow \underline{F}A\rangle\!\rangle(\mathtt{force}\ x)) & \sqsupseteq\sqsubseteq \\ \mathtt{thunk}\ ((\mathtt{force}\ x)) & \sqsupseteq\sqsubseteq \\ x & \end{array}$$

using η for U types and the identity principle for $\langle\!\langle \underline{F}A \nwarrow \underline{F}A\rangle\!\rangle$ (proved analogously to Theorem 72).

2. We use the downcast lemma with $\underline{Y}$ comp type $\vdash \underline{F}U\underline{Y}$ comp type, where $x' : U\underline{B}' \leftarrow \bullet; \mathtt{ret}\ (\langle\!\langle U\underline{B} \swarrow U\underline{B}'\rangle\!\rangle x)$ clearly satisfies typing and monotonicity.

   Finally, for identity extension, we have

$$\begin{array}{rl} x : \underline{B} \leftarrow \bullet; \mathtt{ret}\ (\langle\!\langle \underline{B} \swarrow \underline{B}\rangle\!\rangle x) & \sqsupseteq\sqsubseteq \\ x : \underline{B} \leftarrow \bullet; \mathtt{ret}\ (x) & \sqsupseteq\sqsubseteq \\ \bullet & \end{array}$$

   using the identity principle for $\langle\!\langle \underline{B} \swarrow \underline{B}\rangle\!\rangle$ (proved analogously to Theorem 72) and η for F types.

□

5.3.5 *Upcasts must be Values, Downcasts must be Stacks*

While it may seem like an arbitrary choice to define upcasts as values and downcasts as stacks, rather than the a priori more general definition that upcasts from A to A' are effectful terms $x : A \vdash \underline{F}A'$, which is equivalent to assuming that they are given by a stack upcast $\langle \underline{F}A' \nwarrow \underline{F}A\rangle$ and dually that computations be given by an a priori non-linear term $z : UB' \vdash UB$, which is equivalent to a value downcast $\langle U\underline{B} \swarrow U\underline{B}'\rangle$. We show now that this choice is essentially *forced* upon us, under the mild assumption that certain "ground" up/downcasts are values/stacks. For this section, we define a *ground type*[3] to be generated by the following grammar:

$$G ::= 1 \mid ? \times ? \mid 0 \mid ? + ? \mid U\underline{¿} \qquad \underline{G} ::= ? \to \underline{¿} \mid \top \mid \underline{¿}\ \&\ \underline{¿} \mid \underline{F}?$$

3 In gradual typing, "ground" is used to mean a one-level unrolling of a dynamic type, not first-order data.

Let GTT_G be the fragment of GTT where the only primitive casts are those between ground types and the dynamic types, i.e. the cast terms are restricted to $\langle ? \nwarrow G \rangle V, \langle \underline{F}G \swarrow \underline{F}? \rangle, \langle \underline{G} \swarrow \underline{¿} \rangle E, \langle U\underline{¿} \nwarrow U\underline{G} \rangle E$.

Lemma 81 (Casts are Admissible). *In GTT_G, it is admissible that*

1. *for all $A \sqsubseteq A'$ there is a complex value $\langle\langle A' \nwarrow A \rangle\rangle$ satisfying the universal property of an upcast and a complex stack $\langle\langle \underline{F}A \swarrow \underline{F}A' \rangle\rangle$ satisfying the universal property of a downcast*
2. *for all $\underline{B} \sqsubseteq \underline{B}'$ there is a complex stack $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$ satisfying the universal property of a downcast and a complex value $\langle\langle U\underline{B}' \nwarrow U\underline{B} \rangle\rangle$ satisfying the universal property of an upcast.*

Proof. At the end of this subsection. □

The admissibility theorem is proved by induction over a restricted form of type precision derivations. similar to the one developed in Figure 3.

Definition 82 (Ground type precision). Let $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq' \underline{B}'$ be the relations defined by the rules in Figure 5.3 with the axioms $A \sqsubseteq ?$ and $\underline{B} \sqsubseteq \underline{¿}$ restricted to ground types—i.e., replaced by $G \sqsubseteq ?$ and $\underline{G} \sqsubseteq \underline{¿}$.

Lemma 83. *For any type A, $A \sqsubseteq' ?$. For any type $\underline{B}$, $\underline{B} \sqsubseteq' \underline{¿}$.*

Proof. By induction on the type. For example, in the case for $A_1 + A_2$, we have by the inductive hypothesis $A_1 \sqsubseteq' ?$ and $A_2 \sqsubseteq' ?$, so $A_1 + A_2 \sqsubseteq' ? + ? \sqsubseteq ?$ by congruence and transitivity, because $? + ?$ is ground. In the case for $\underline{F}A$, we have $A \sqsubseteq ?$ by the inductive hypothesis, so $\underline{F}A \sqsubseteq \underline{F}? \sqsubseteq \underline{¿}$. □

Lemma 84 ($\sqsubseteq$ and $\sqsubseteq'$ agree). *$A \sqsubseteq A'$ iff $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq \underline{B}'$ iff $\underline{B} \sqsubseteq' \underline{B}'$*

Proof. The "if" direction is immediate by induction because every rule of $\sqsubseteq'$ is a rule of $\sqsubseteq$. To show $\sqsubseteq$ is contained in $\sqsubseteq'$, we do induction on the derivation of $\sqsubseteq$, where every rule is true for $\sqsubseteq'$, except $A \sqsubseteq ?$ and $\underline{B} \sqsubseteq \underline{¿}$, and for these, we use Lemma 83. □

Proof. To streamline the exposition above, we stated Theorem 72, Theorem 75 Theorem 76 as showing that the "definitions" of each cast are equiprecise with the cast that is a priori postulated to exist (e.g. $\langle A'' \nwarrow A \rangle \sqsupseteq\sqsubseteq \langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle$). However, the proofs factor through Theorem 71 and Lemma 79 and Lemma 80, which show directly that the right-hand sides have the desired universal property—i.e. the stipulation that some cast with the correct universal property exists is not used in the proof that the implementation has the desired universal property. Moreover, the proofs given do not rely on any axioms of GTT besides the universal properties of the "smaller" casts

used in the definition and the $\beta\eta$ rules for the relevant types. So these proofs can be used as the inductive steps here, in GTT_G.

By induction on type precision $A \sqsubseteq' A'$ and $\underline{B} \sqsubseteq' \underline{B}'$.

(We chose not to make this more explicit above, because we believe the equational description in a language with all casts is a clearer description of the results, because it avoids needing to hypothesize terms that behave as the smaller casts in each case.)

We show a few representative cases:

In the cases for $G \sqsubseteq ?$ or $\underline{G} \sqsubseteq ¿$, we have assumed appropriate casts $\langle ? \nwarrow G \rangle$ and $\langle \underline{F}G \swarrow \underline{F}? \rangle$ and $\langle \underline{G} \swarrow ¿ \rangle$ and $\langle U¿ \nwarrow U\underline{G} \rangle$.

In the case for identity $A \sqsubseteq A$, we need to show that there is an upcast $\langle\langle A \nwarrow A \rangle\rangle$ and a downcast $\langle\langle \underline{F}A \swarrow \underline{F}A \rangle\rangle$ The proof of Theorem 72 shows that the identity value and stack have the correct universal property.

In the case where type precision was concluded by transitivity between $A \sqsubseteq A'$ and $A' \sqsubseteq A''$, by the inductive hypotheses we get upcasts $\langle\langle A' \nwarrow A \rangle\rangle$ and $\langle\langle A'' \nwarrow A' \rangle\rangle$, and the proof of Theorem 72 shows that defining $\langle\langle A'' \nwarrow A \rangle\rangle$ to be $\langle\langle A'' \nwarrow A' \rangle\rangle\langle\langle A' \nwarrow A \rangle\rangle$ has the correct universal property. For the downcast, we get $\langle\langle \underline{F}A \swarrow \underline{F}A' \rangle\rangle$ and $\langle\langle \underline{F}A' \swarrow \underline{F}A'' \rangle\rangle$ by the inductive hypotheses, and the proof of Theorem 72 shows that their composition has the correct universal property.

In the case where type precision was concluded by the congruence rule for $A_1 + A_2 \sqsubseteq A_1' + A_2'$ from $A_i \sqsubseteq A_i'$, we have upcasts $\langle\langle A_i' \nwarrow A_i \rangle\rangle$ and downcasts $\langle\langle \underline{F}A_i \swarrow \underline{F}A_i' \rangle\rangle$ by the inductive hypothesis, and the proof of Theorem 72 shows that the definitions given there have the desired universal property.

In the case where type precision was concluded by the congruence rule for $\underline{F}A \sqsubseteq \underline{F}A'$ from $A \sqsubseteq A'$, we obtain by induction an *upcast* $A \sqsubseteq A'$ and a downcast $\langle\langle \underline{F}A \swarrow \underline{F}A' \rangle\rangle$. We need a *downcast* $\langle\langle \underline{F}A \swarrow FA' \rangle\rangle$, which we have, and an *upcast* $\langle\langle U\underline{F}A \swarrow U\underline{F}A' \rangle\rangle$, which is constructed as in Theorem 76. □

As discussed in §5.2.2.2, rather than an upcast being a complex value $x : A \vdash \langle A' \nwarrow A \rangle x : A'$, an a priori more general type would be a stack $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$, which allows the upcast to perform effects; dually, an a priori more general type for a downcast $\bullet : \underline{B}' \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}$ would be a value $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$, which allows the downcast to ignore its argument. The following shows that in GTT_G, if we postulate such stack upcasts/value downcasts as originally suggested in §5.2.2.2, then in fact these casts *must* be equal to the action of $U/\underline{F}$ on some value upcasts/stack downcasts, so the potential for effectfulness/non-linearity affords no additional flexibility.

Theorem 85 (Upcasts are Necessarily Values, Downcasts are Necessarily Stacks)**.** *Suppose we extend GTT_G with the following postulated stack*

upcasts and value downcasts (in the sense of Definition 69): For every type precision $A \sqsubseteq A'$, there is a stack upcast $\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet : \underline{F}A'$, and for every $\underline{B} \sqsubseteq \underline{B}'$, there is a complex value downcast $x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x : U\underline{B}$.

Then there exists a value upcast $\langle\langle A' \nwarrow A \rangle\rangle$ and a stack downcast $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$ such that

$$\bullet : \underline{F}A \vdash \langle \underline{F}A' \nwarrow \underline{F}A \rangle \bullet \sqsupseteq\sqsubseteq (x : A \leftarrow \bullet; \mathtt{ret}\ (\langle\langle A' \nwarrow A \rangle\rangle x))$$
$$x : U\underline{B}' \vdash \langle U\underline{B} \swarrow U\underline{B}' \rangle x \sqsupseteq\sqsubseteq (\mathtt{thunk}\ (\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle (\mathtt{force}\ x)))$$

Proof. Lemma 81 constructs $\langle\langle A' \nwarrow A \rangle\rangle$ and $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$, so the proof of Theorem 76 (which really works for any $\langle\langle A' \nwarrow A \rangle\rangle$ and $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$ with the correct universal properties, not only the postulated casts) implies that the right-hand sides of the above equations are stack upcasts and value downcasts of the appropriate type. Since stack upcasts/value downcasts are unique by an argument analogous to Theorem 71, the postulated casts must be equal to these. □

Indeed, the following a priori even more general assumption provides no more flexibility:

Theorem 86 (Upcasts are Necessarily Values, Downcasts are Necessarily Stacks II)**.** *Suppose we extend GTT_G only with postulated monadic upcasts $x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A \rangle x : U\underline{F}A'$ for every $A \sqsubseteq A'$ and comonadic downcasts $\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}' \rangle \bullet : \underline{F}U\underline{B}$ for every $\underline{B} \sqsubseteq \underline{B}'$.*

Then there exists a value upcast $\langle\langle A' \nwarrow A \rangle\rangle$ such that

$$x : U\underline{F}A \vdash \langle U\underline{F}A' \nwarrow U\underline{F}A \rangle x \sqsupseteq\sqsubseteq \mathtt{thunk}\ (x : A \leftarrow \mathtt{force}\ x; \mathtt{ret}\ (\langle\langle A' \nwarrow A \rangle\rangle x))$$

and a stack downcast $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$ such that

$$\bullet : \underline{F}U\underline{B}' \vdash \langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}' \rangle \bullet \sqsupseteq\sqsubseteq x' : U\underline{B}' \leftarrow \bullet; \mathtt{ret}\ (\mathtt{thunk}\ (\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle (\mathtt{force}\ x)))$$

In CBV terms, the monadic upcast is like an upcast from A to A' taking having type $(1 \to A) \to A'$, i.e. it takes a thunked effectful computation of an A as input and produces an effectful computation of an A'.

Proof. Again, Lemma 81 constructs $\langle\langle A' \nwarrow A \rangle\rangle$ and $\langle\langle \underline{B} \swarrow \underline{B}' \rangle\rangle$, so the proof of Theorem 76 gives the result. □

5.3.6 *Equiprecision and Isomorphism*

Since we can relate types by precision ($\sqsubseteq$) and terms (values, stacks, computations), there are several different notions of equivalence of types. The most useful of these are *equiprecision* of types ($\sqsupseteq\sqsubseteq$) and

isomorphism of types. In CBPV, the appropriate definition of isomorphism is *pure* value isomorphism between value types and *linear* stack isomorphism between computation types.

Definition 87 (Isomorphism)**.**

1. We write $A \cong_v A'$ for a *value isomorphism between A and A'*, which consists of two complex values $x : A \vdash V' : A'$ and $x' : A' \vdash V : A$ such that $x : A \vdash V[V'/x'] \sqsupseteq\sqsubseteq x : A$ and $x' : A' \vdash V'[V/x] \sqsupseteq\sqsubseteq x' : A'$.

2. We write $\underline{B} \cong_c \underline{B}'$ for a *computation isomorphism between $\underline{B}$ and $\underline{B}'$*, which consists of two complex stacks $\bullet : \underline{B} \vdash S' : \underline{B}'$ and $\bullet' : \underline{B}' \vdash S : \underline{B}$ such that $\bullet : \underline{B} \vdash S[S'/x'] \sqsupseteq\sqsubseteq \bullet : \underline{B}$ and $\bullet' : \underline{B}' \vdash S'[S/\bullet] \sqsupseteq\sqsubseteq \bullet' : \underline{B}'$.

Note that value and computation isomorphisms are a stronger condition than isomorphism in call-by-value and call-by-name. An isomorphism in call-by-value between types A and A' corresponds to a computation isomorphism $\underline{F}A \cong_c \underline{F}A'$, and dually a call-by-name isomorphism between $\underline{B}$ and $\underline{B}'$ corresponds to a value isomorphism $U\underline{B} \cong_v U\underline{B}'$ [46].

As discussed in our previous work on call-by-name GTT New and Licata [58, 59], equiprecision is stronger than isomorphism: isomorphism says that the "elements" of the types are in one-to-one correspondence, but equiprecision says additionally that those "elements" are represented in the same way at the dynamic type. To see this formally, first observe:

Theorem 88 (Equiprecision implies Isomorphism)**.** *1. If $A \sqsupseteq\sqsubseteq A'$, then $\langle A' \nwarrow A \rangle$ and $\langle A \nwarrow A' \rangle$ form a value isomorphism $A \cong_v A'$.*

2. If $\underline{B} \sqsupseteq\sqsubseteq \underline{B}'$, then $\langle \underline{B} \swarrow \underline{B}' \rangle$ and $\langle \underline{B}' \swarrow \underline{B} \rangle$ form a computation isomorphism $\underline{B} \cong_c \underline{B}'$.

Proof. 1. We have upcasts $x : A \vdash \langle A' \nwarrow A \rangle x : A'$ and $x' : A' \vdash \langle A \nwarrow A' \rangle x' : A$. For the composites, to show $x : A \vdash \langle A \nwarrow A' \rangle \langle A' \nwarrow A \rangle x \sqsubseteq x$ we apply upcast left twice, and conclude $x \sqsubseteq x$ by assumption. To show, $x : A \vdash x \sqsubseteq \langle A \nwarrow A' \rangle \langle A' \nwarrow A \rangle x$, we have $x : A \vdash x \sqsubseteq \langle A' \nwarrow A \rangle x : A \sqsubseteq A'$ by upcast right, and therefore $x : A \vdash x \sqsubseteq \langle A \nwarrow A' \rangle \langle A' \nwarrow A \rangle x : A \sqsubseteq A$ again by upcast right. The other composite is the same proof with A and A' swapped.

2. We have downcasts $\bullet : \underline{B} \vdash \langle \underline{B} \swarrow \underline{B}' \rangle \bullet : \underline{B}'$ and $\bullet : \underline{B}' \vdash \langle \underline{B}' \swarrow \underline{B} \rangle \bullet : \underline{B}$.

   For the composites, to show $\bullet : \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B}' \swarrow \underline{B} \rangle \langle \underline{B} \swarrow \underline{B}' \rangle \bullet$, we apply downcast right twice, and conclude $\bullet \sqsubseteq \bullet$. For $\langle \underline{B}' \swarrow \underline{B} \rangle \langle \underline{B} \swarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet$, we first have $\langle \underline{B} \swarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$ by downcast left, and then the result by another application of

downcast left. The other composite is the same proof with $\underline{B}$ and $\underline{B}'$ swapped.

□

On the other hand, we should not expect that isomorphism implies equiprecision, since there are many non-trivial isomorphisms that will have different encodings in the dynamic type. For instance $U\underline{B} \times 1 \cong_v A$ but the former will typically be represented as a cons of a thunk and a dummy value. For another example, $U(\underline{B}_1 \,\&\, \underline{B}_2) \cong_v U\underline{B}_1 \,\&\, U\underline{B}_2$ but the former would typically be represented as a single closure that can be called with either of two methods, whereas the latter will be a cons of two closures.

5.3.7 *Most Precise Types*

Though it is common in gradually typed surface languages to have a *most* dynamic type in the form of the dynamic type ?, it is less common to have a *least* dynamic type $\bot$. Having a least dynamic type causes issues with certain definitions. For instance sometimes the type consistency relation $A \sim A'$ is defined as existence of a type more precise than each: $\exists A_l.A_l \sqsubseteq A \wedge A_l \sqsubseteq A'$, but this definition would be trivial given the presence of a most precise type.

We consider here the *semantic* consequences of having a least dynamic value type $\bot_{\mathsf{v}}$ or computation type $\bot_{\mathsf{c}}$, which are quite mild. In the case of the most precise value type $\bot_{\mathsf{v}}$, we have a pure value $x : \bot_{\mathsf{v}} \vdash \langle A \leftharpoonup \bot_{\mathsf{v}} \rangle x : A$ for every value type A. This suggests that the empty type 0 is a candidate to be $\bot_{\mathsf{v}}$, and in fact we can show the two are isomorphic.

To prove this we first recall some general facts about the empty type, in category theoretic terms that it is a *strictly initial* object.

Lemma 89 ((Strictly) Initial Object)**.** *All of the following are true.*

1. *For all (value or computation) types T, there exists a unique expression $x : 0 \vdash E : T$. In category-theoretic terms, 0 is initial in the category of value types and values.*

2. *For all $\underline{B}$, there exists a unique stack $\bullet : \underline{F}0 \vdash S : \underline{B}$. In category-theoretic terms, $\underline{F}0$ is initial in the category of computation types and stacks.*

3. *Suppose there is a type A with a complex value $x : A \vdash V : 0$. Then V is an isomorphism $A \cong_v 0$. In category-theoretic terms, 0 is* strictly *initial.*

Proof. 1. Take E to be $x : 0 \vdash \texttt{abort}\ x : T$. Given any E', we have $E \sqsupseteq\sqsubseteq E'$ by the η principle for 0.

2. Take S to be $\bullet : \underline{F}0 \vdash x \leftarrow \bullet; \mathtt{abort}\ x : \underline{B}$. Given another S', by the η principle for F types, $S' \sqsupseteq\sqsubseteq x \leftarrow \bullet; S'[\mathtt{ret}\ x]$. By congruence, to show $S \sqsupseteq\sqsubseteq S'$, it suffices to show $x : 0 \vdash \mathtt{abort}\ x \sqsupseteq\sqsubseteq S[\mathtt{ret}\ x] : \underline{B}$, which is an instance of the previous part.

3. We have $y : 0 \vdash \mathtt{abort}\ y : A$. The composite $y : 0 \vdash V[\mathtt{abort}\ y/x] : 0$ is equiprecise with y by the η principle for 0, which says that any two complex values with domain 0 are equal.

   The composite $x : A \vdash \mathtt{abort}\ V : A$ is equiprecise with x, because

   $$x : A, y : A, z : 0 \vdash x \sqsupseteq\sqsubseteq \mathtt{abort}\ z \sqsupseteq\sqsubseteq y : A$$

   where the first is by η with $x : A, y : A, z : 0 \vdash E[z] := x : A$ and the second with $x : 0, y : 0 \vdash E[z] := y : A$ (this depends on the fact that 0 is "distributive", i.e. $\Gamma, x : 0$ has the universal property of 0). Substituting $\mathtt{abort}\ V$ for y and V for z, we have $\mathtt{abort}\ V \sqsupseteq\sqsubseteq x$.

□

Note however that we cannot prove that $\underline{F}0$ is *strictly* initial in the category of stacks.

With this lemma in hand, we can show that $\bot_{\mathsf{v}}$ must be value-isomorphic to 0:

Theorem 90 (Most Precise Value Type)**.** *If $\bot_{\mathsf{v}}$ is a type such that $\bot_{\mathsf{v}} \sqsubseteq A$ for all A, then in GTT with 0, $\bot_{\mathsf{v}} \cong_v 0$.*

Proof. We have the upcast $x : \bot_{\mathsf{v}} \vdash \langle 0 \nwarrow \bot_{\mathsf{v}} \rangle x : 0$, so Lemma 89 gives the result. □

However, note that unless we already know there is an empty type 0, we see no way to prove that $\bot_{\mathsf{v}}$ is initial in that all terms $x : \bot_{\mathsf{v}} \vdash M$ are equivalent.

Thinking dually, a most precise computation type would have a linear stack $\bullet : \underline{B} \vdash \langle \bot_{\mathsf{c}} \swarrow \underline{B} \rangle \bullet : \bot_{\mathsf{c}}$ for every computation type $\underline{B}$, so an obvious candidate would be the lazy unit $\top$, the dual of the empty type. However, the duality here is not perfect and we will only be able to prove the weaker fact that $U\top$ and $U\bot_{\mathsf{c}}$ are isomorphic.

To prove this, we first recall the defining property of $\top$, that it is in category-theoretic terms a *terminal object*, but not provably a *strictly* terminal object, breaking the precise duality with 0.

Lemma 91 (Terminal Objects)**.**

1. *For any computation type $\underline{B}$ and context Γ, there exists a unique stack $\Gamma \mid \bullet : \underline{B} \vdash S : \top$, i.e., $\top$ is a terminal object in the category of computation types and stacks.*

2. *In any context Γ, there exists a unique complex value $V : U\top$, i.e., $U\top$ is a terminal object in the category of value types and values.*

3. *(In any context* Γ*,) there exists a unique complex value* $V : 1$*, i.e.,* 1 *is also a terminal object.*

4. $U\top \cong_v 1$

Proof. 1. Take $S = \{\}$. The η rule for $\top$, $\bullet : \top \vdash \bullet \sqsupseteq\sqsubseteq \{\} : \top$, under the substitution of $\bullet : \underline{B} \vdash S : \top$, gives $S \sqsupseteq\sqsubseteq \{\}[S/\bullet] = \{\}$.

2. Take $V = \texttt{thunk}\ \{\}$. We have $x : U\top \vdash x \sqsupseteq\sqsubseteq \texttt{thunk force}\ x \sqsupseteq\sqsubseteq \texttt{thunk}\ \{\} : U\top$ by the η rules for U and $\top$.

3. Take $V = ()$. By η for 1 with $x : 1 \vdash E[x] := () : 1$, we have $x : 1 \vdash () \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ (). : 1$. By η fro 1 with $x : 1 \vdash E[x] := x : 1$, we have $x : 1 \vdash x \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ ()$.. Therefore $x : 1 \vdash x \sqsupseteq\sqsubseteq () : 1$.

4. We have maps $x : U\top \vdash () : 1$ and $x : 1 \vdash \texttt{thunk}\ \{\} : U\top$. The composite on 1 is the identity by the previous part. The composite on $\top$ is the identity by part (2).

□

Note that we cannot show that $\top$ is strictly terminal. Next, we can show that $U\bot_c$ is isomorphic to $U\top$.

Theorem 92 (Most Precise Computation Type)**.** *If* $\bot_c$ *is a type such that* $\bot_c \sqsubseteq \underline{B}$ *for all* $\underline{B}$*, and we have a terminal computation type* $\top$*, then* $U\bot_c \cong_v U\top$.

Proof. First, though we can define stacks $\bullet : \top\langle \bot_c \swarrow \top\rangle \bullet : \bot_c$ and $\bullet : \bot_c \vdash \{\} : \top$, we can only prove one direction of the isomorphism:

$$\{\}[\langle \bot_c \swarrow \top\rangle \bullet] \sqsupseteq\sqsubseteq \{\} \sqsupseteq\sqsubseteq \bullet$$

Since $\top$ is not a strict terminal object, the dual of the above argument does not give the other property of a stack isomorphism $\bot_c \cong_c \top$.

On the other hand, we can define values

$$x : U\bot_c \vdash \langle U\top \nwarrow U\bot_c\rangle x : U\top$$

$$y : U\top \vdash \langle\!\langle U\bot_c \swarrow U\top\rangle\!\rangle y : U\bot_c$$

And these do exhibit the isomorphism $U\bot_c \cong_v U\top$. First, by the retract axiom

$$x : U\bot_c \vdash \langle\!\langle U\bot_c \swarrow U\top\rangle\!\rangle \langle U\top \nwarrow U\bot_c\rangle x \sqsupseteq\sqsubseteq x : U\bot_c$$

and the opposite composite

$$y : U\top \vdash \langle U\top \nwarrow U\bot_c\rangle \langle\!\langle U\bot_c \swarrow U\top\rangle\!\rangle : U\top$$

is the identity by uniqueness for $U\top$ (Lemma 91). □

Given these two Theorems 90, 92, it is then sensible to ask what are the consequences of *defining* 0 and $\top$ to be most precise types. If this is the case then, like in Section , we can derive what the behavior of their casts would be.

Theorem 93. *If* $0 \sqsubseteq A$, *then*

$$\langle A \nwarrow 0\rangle z \sqsupseteq\sqsubseteq \mathtt{absurd}\ z \qquad \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet \sqsupseteq\sqsubseteq \_ \leftarrow \bullet; \mho$$

If $\top \sqsubseteq \underline{B}$, *then*

$$\langle \top \swarrow \underline{B}\rangle \bullet \sqsupseteq\sqsubseteq \{\} \qquad \langle U\underline{B} \nwarrow U\top\rangle u \sqsupseteq\sqsubseteq \mathtt{thunk}\ \mho$$

Proof. 1. $x : 0 \vdash \langle A \nwarrow 0\rangle x \sqsupseteq\sqsubseteq \mathtt{abort}\ x : A$ is immediate by η for 0.

2. First, to show $\bullet : \underline{F}A \vdash \_ \leftarrow \bullet; \mho \sqsubseteq \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet$, we can η-expand the right-hand side into $x : A \leftarrow \bullet; \langle \underline{F}0 \swarrow \underline{F}A\rangle \mathtt{ret}\ x$, at which point the result follows by congruence and the fact that type error is minimal, so $\mho \sqsubseteq \langle \underline{F}0 \swarrow \underline{F}A\rangle \mathtt{ret}\ x$.

   Second, to show $\bullet : \underline{F}A \vdash \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet \sqsubseteq \_ \leftarrow \bullet; \mho$, we can η-expand the left-hand side to $\bullet : \underline{F}A \vdash y \leftarrow \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet; \mathtt{ret}\ y$, so we need to show

   $$\bullet : \underline{F}A \vdash y : 0 \leftarrow \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet; \mathtt{ret}\ y \sqsubseteq y' : A \leftarrow \bullet; \mho : \underline{F}0$$

   We apply congruence, with $\bullet : \underline{F}A \vdash \langle \underline{F}0 \swarrow \underline{F}A\rangle \bullet \sqsubseteq \bullet : 0 \sqsubseteq A$ by the universal property of downcasts in the first premise, so it suffices to show

   $$y \sqsubseteq y' : 0 \sqsubseteq A \vdash \mathtt{ret}\ y \sqsubseteq \mho_{\underline{F}0} : \underline{F}0$$

   By transitivity with $y \sqsubseteq y' : 0 \sqsubseteq A \vdash \mho_{\underline{F}0} \sqsubseteq \mho_{\underline{F}0} : \underline{F}0 \sqsubseteq \underline{F}0$, it suffices to show

   $$y \sqsubseteq y : 0 \sqsubseteq 0 \vdash \mathtt{ret}\ y \sqsubseteq \mho_{\underline{F}0} : \underline{F}0$$

   But now both sides are maps out of 0, and therefore equal by Lemma 89.

3. The downcast is immediate by η for $\top$, Lemma 91.

4. First,

$$u : U\top \vdash \mathtt{thunk}\ \mho \sqsubseteq \mathtt{thunk}\ (\mathtt{force}\ (\langle U\underline{B} \nwarrow U\top\rangle u)) \sqsupseteq\sqsubseteq \langle U\underline{B} \nwarrow U\top\rangle u : U\underline{B}$$

   by congruence, η for U, and the fact that error is minimal. Conversely, to show

   $$u : U\top \vdash \langle U\underline{B} \nwarrow U\top\rangle u \sqsubseteq \mathtt{thunk}\ \mho : U\underline{B}$$

it suffices to show

$$u : U\top \vdash u \sqsubseteq \mathtt{thunk}\ \mho_{\underline{B}} : U\top \sqsubseteq U\underline{B}$$

by the universal property of an upcast. By Lemma 91, any two elements of $U\top$ are equiprecise, so in particular $u \sqsupseteq\sqsubseteq \mathtt{thunk}\ \mho_{\top}$, at which point congruence for thunk and $\mho_{\top} \sqsubseteq \mho_{\underline{B}} : \top \sqsubseteq \underline{B}$ gives the result.

□

5.4 DISCUSSION AND RELATED WORK

In this chapter, we have given a logic for reasoning about gradual programs in a mixed call-by-value/call-by-name language, shown that the axioms uniquely determine almost all of the contract translation implementing runtime casts. In Chapter 7, we will also that the axiomatics is sound for contextual equivalence/approximation in an operational model.

In immediate future work, we believe it is straightforward to add inductive/coinductive types and obtain similar unique cast implementation theorems such as

$$\langle \mathtt{list}(A') \leftarrow \mathtt{list}(A) \rangle \sqsupseteq\sqsubseteq \mathtt{map}\langle A' \leftarrow A \rangle.$$

The upcast/downcast lemmas () should apply immediately. Additionally, since more efficient cast implementations such as optimized cast calculi (the lazy variant in Herman, Tomb, and Flanagan [39]) and threesome casts [71], are equivalent to the lazy contract semantics, they should also be models of GTT, and if so we could use GTT to reason about program transformations and optimizations in them.

APPLICABILITY OF CAST UNIQUENESS PRINCIPLES The cast uniqueness principles given in Theorem 75 are theorems in the formal logic of Gradual Type Theory, and so there is a question of to what languages the theorem applies. The theorem applies to any *model* of gradual type theory, such as the models we have constructed using call-by-push-value given in Chapter 7. We conjecture that simple call-by-value and call-by-name gradual languages are also models of GTT, by extending the translation of call-by-push-value into call-by-value and call-by-name in the appendix of Levy's monograph [45]. In order for the theorem to apply, the language must validate an appropriate version of the η principles for the types. So for example, a call-by-value language that has reference equality of functions does *not* validate even the value-restricted η law for functions, and so the case for functions does not apply. It is a well-known issue that in the presence of pointer equality of functions, the lazy semantics of function casts is not compatible with the graduality property, and our uniqueness theorem provides a different perspective on this phenomenon [26, 75,

76]. However, we note that the cases of the uniqueness theorem for each type connective are completely *modular*: they rely only on the specification of casts and the β, η principles for the particular connective, and not on the presence of any other types, even the dynamic types. So even if a call-by-value language may have reference equality functions, if it has the η principle for strict pairs, then the pair cast must be that of Theorem 75.

Next, we consider the applicability to non-eager languages. Analogous to call-by-value, our uniqueness principle should apply to simple *call-by-name* gradual languages, where full η equality for functions is satisfied, but η equality for booleans and strict pairs requires a "stack restriction" dual to the value restriction for call-by-value function η. We are not aware of any call-by-name gradual languages, but there is considerable work on *contracts* for non-eager languages, especially Haskell [40, 91]. However, we note that Haskell is *not* a call-by-name language in our sense for two reasons. First, Haskell uses call-by-need evaluation where results of computations are memoized. However, when only considering Haskell's effects (error and divergence), this difference is not observable so this is not the main obstacle. The bigger difference between Haskell and call-by-name is that Haskell supports a `seq` operation that enables the programmer to force evaluation of a term to a value. This means Haskell violates the function η principle because Ω will cause divergence under `seq` , whereas $\lambda x.\Omega$ will not. This is a crucial feature of Haskell and is a major source of differences between implementations of lazy contracts, as noted in Degen, Thiemann, and Wehr [17]. We can understand this difference by using a different translation into call-by-push-value: what Levy calls the "lazy paradigm", as opposed to call-by-name [45]. Simply put, connectives are interpreted as in call-by-value, but with the addition of extra thunks UF, so for instance the lazy function type $A \to B$ is interpreted as $UFU(UFA \to FB)$ and the extra UFU here is what causes the failure of the call-by-name η principle. With this embedding and the uniqueness theorem, GTT produces a definition for lazy casts, and the definition matches the work of Xu, Peyton Jones, and Claessen [91] when restricting to non-dependent contracts.

COMPARING SOUNDNESS PRINCIPLES FOR CAST SEMANTICS Greenman and Felleisen [33] gives a spectrum of differing syntactic type soundness theorems for different semantics of gradual typing. Our work here is complementary, showing that certain program equivalences can only be achieved by certain cast semantics.

Degen, Thiemann, and Wehr [17] give an analysis of different cast semantics for contracts in lazy languages, specifically based on Haskell, i.e., call-by-need with `seq` . They propose two properties "meaning preservation" and "completeness" that they show are incompatible and identify which contract semantics for a lazy language satisfy

which of the properties. The meaning preservation property is closely related to graduality: it says that evaluating a term with a contract either produces blame or has the same observable effect as running the term without the contract. Meaning preservation rules out overly strict contract systems that force (possibly diverging) thunks that wouldn't be forced in a non-contracted term. Completeness, on the other hand, requires that when a contract is attached to a value that it is *deeply* checked. The two properties are incompatible because, for instance, a pair of a diverging term and a value can't be deeply checked without causing the entire program to diverge. Using Levy's embedding of the lazy paradigm into call-by-push-value their incompatibility theorem should be a consequence of our main theorem in the following sense. We showed that any contract semantics departing from the implementation in Theorem 75 must violate η or graduality. Their completeness property is inherently eager, and so must be different from the semantics GTT would provide, so either the restricted η or graduality fails. However, since they are defining contracts within the language, they satisfy the restricted η principle provided by the language, and so it must be graduality, and therefore meaning preservation that fails.

AXIOMATIC CASTS Henglein's work on dynamic typing also uses an axiomatic semantics of casts, but axiomatizes behavior of casts at each type directly whereas we give a uniform definition of all casts and derive implementations for each type [38]. Because of this, the theorems proven in that paper are more closely related to our model construction in Chapter 7. More specifically, many of the properties of casts needed to prove Theorem 128 have direct analogues in Henglein's work, such as the coherence theorems. Finally, we note that our assumption of compositionality, i.e., that all casts can be decomposed into an upcast followed by a downcast, is based on Henglein's analysis, where it was proven to hold in his coercion calculus.

GRADUAL TYPING FRAMEWORKS In this work we have applied a method of "gradualizing" axiomatic type theories by adding in precision orderings and adding dynamic types, casts and errors by axioms related to the precision orderings. This is similar in spirit to two recent frameworks for designing gradual languages: Abstracting Gradual Typing (AGT) [29] and the Gradualizer [13, 14]. All of these approaches start with a typed language and construct a related gradual language. A major difference between our approach and those is that our work is based on axiomatic semantics and so we take into account the equality principles of the typed language, whereas Gradualizer is based on the typing and operational semantics and AGT is based on the type safety proof of the typed language. Furthermore, our approach produces not just a single language, but also an axiomatization of the structure of gradual typing and so we can prove results about many

languages by proving theorems in GTT. The downside to this is that our approach doesn't directly provide an operational semantics for the gradual language, whereas for AGT this is a semi-mechanical process and for Gradualizer, completely automated. Finally, we note that AGT produces the "eager" semantics for function types, and it is not clear how to modify the AGT methodology to reproduce the lazy semantics that GTT provides.

BLAME We do not give a treatment of runtime blame reporting, but we argue that the observation that upcasts are thunkable and downcasts are linear is directly related to blame soundness [81, 88] in that if an upcast were *not* thunkable, it should raise positive blame and if a downcast were *not* linear, it should raise negative blame. First, consider a potentially effectful stack upcast of the form $\langle \underline{F}A' \nwarrow \underline{F}A \rangle$. If it is not thunkable, then in our logical relation this would mean there is a value $V : A$ such that $\langle \underline{F}A' \nwarrow \underline{F}A \rangle(\mathtt{ret}\ V)$ performs some effect. Since the only observable effects for casts are dynamic type errors, $\langle \underline{F}A' \nwarrow \underline{F}A \rangle(\mathtt{ret}\ V) \mapsto \mathcal{O}$, and we must decide whether the positive party or negative party is at fault. However, since this is call-by-value evaluation, this error happens unconditionally on the continuation, so the continuation never had a chance to behave in such a way as to prevent blame, and so we must blame the positive party.

Dually, consider a value downcast of the form $\langle U\underline{B} \swarrow U\underline{B}' \rangle$. If it is not linear, that would mean it forces its $U\underline{B}'$ input either never or more than once. Since downcasts should refine their inputs, it is not possible for the downcast to use the argument twice, since e.g., printing twice does not refine printing once. So if the cast is not linear, that means it fails without ever forcing its input, in which case it knows nothing about the positive party and so must blame the negative party. In future work, we plan to investigate extensions of GTT with more than one $\mathcal{O}$ with different blame labels, and an axiomatic account of a blame-aware observational equivalence.

DENOTATIONAL AND CATEGORY-THEORETIC MODELS We have presented certain concrete models of GTT using ordered CBPV with errors, in order to efficiently arrive at a concrete operational interpretation. It may be of interest to develop a more general notion of model of GTT for which we can prove soundness and completeness theorems, as in New and Licata [58]. A model would be a strong adjunction between double categories where one of the double categories has all "companions" and the other has all "conjoints", corresponding to our upcasts and downcasts. Then the contract translation should be a construction that takes a strong adjunction between 2-categories and makes a strong adjunction between double categories where the ep pairs are "Kleisli" ep pairs: the upcast is has a right adjoint, but only

in the Kleisli category and vice-versa the downcast has a left adjoint in the co-Kleisli category.

Furthermore, the ordered CBPV with errors should also have a sound and complete notion of model, and so our contract translation should have a semantic analogue as well.

GRADUAL SESSION TYPES Gradual session types [41] share some similarities to GTT, in that there are two sorts of types (values and sessions) with a dynamic value type and a dynamic session type. However, their language is not *polarized* in the same way as CBPV, so there is not likely an analogue between our upcasts always being between value types and downcasts always being between computation types. Instead, we might reconstruct this in a polarized session type language [62]. The two dynamic types would then be the "universal sender" and "universal receiver" session types.

DYNAMICALLY TYPED CALL-BY-PUSH-VALUE Our interpretation of the dynamic types in call-by-push-value suggests a design for a Scheme-like language with a value and computation distinction. This may be of interest for designing an extension of Typed Racket that efficiently supports CBN or a Scheme-like language with codata types. While the definition of the dynamic computation type by a lazy product may look strange, we argue that it is no stranger than the use of its dual, the sum type, in the definition of the dynamic value type. That is, in a truly dynamically typed language, we would not think of the dynamic type as being built out of some sum type construction, but rather that it is the *union* of all of the ground value types, and the union happens to be a *disjoint* union and so we can model it as a sum type. In the dual, we don't think of the computation dynamic type as a *product*, but instead as the *intersection* of the ground computation types. Thinking of the type as unfolding:

$$\underline{¿} = \underline{F}\underline{¿} \wedge (? \to \underline{F}?) \wedge (? \to ? \to \underline{F}?) \wedge \cdots$$

This says that a dynamically typed computation is one that can be invoked with any finite number of arguments on the stack, a fairly accurate model of implementations of Scheme that pass multiple arguments on the stack.

DEPENDENT CONTRACT CHECKING We also plan to explore using GTT's specification of casts in a dependently typed setting, building on work using Galois connections for casts between dependent types [16, 20], and work on effectful dependent types based a CBPV-like judgement structure [1, 61].

6

FROM GTT TO EVALUATION ORDERS

To show how the call-by-push-value based GTT helps us to understand more common evaluation orders, we show how the single unified theory of GTT produces cast semantics for 3 different evaluation orders: call-by-value, call-by-name and "lazy" semantics. For syntax, we will use the same cast calculus as presented in Chapter 2, §2.1.

For each evaluation order, we will give an operational semantics of the cast calculus and an elaboration to GTT based on Levy's original CBPV translations [45], extended to gradual typing using the principle that any cast is equivalent to casting up to dynamic and then down. Then we will show that in each evaluation order, if M reduces to M' ($M \mapsto M'$), then $[\![M]\!] \sqsupseteq\sqsubseteq [\![M']\!]$ in GTT[1], showing that the reduction is justified from the principles GTT codifies.

The call-by-value evaluation order is the same one we showed in Chapter 2, where each type comes with a notion of value and terms (possibly) reduce to values. Call-by-name evaluation order is based on traditional λ calculus, which validates the strong η principle for functions $\lambda x.Mx \cong M$ even in the presence of effects. In particular, there is no way to tell if a term of function type "terminates" in that a diverging term of function type is equivalent to $\lambda x.\Omega\ x$, so there is no non-trivial, observable notion of value for most types. The third evaluation order, what Levy calls "lazy", is something of a compromise between the two evaluation orders. Like call-by-name, function arguments are passed as unevaluated thunks, but like call-by-value, terms reduce to values and the difference between a diverging term Ω of function type and a function that always diverges $\lambda x.\Omega$ is observable using some kind of "strict sequencing" operation. In the language Haskell, this strict sequencing is given by the construct "seq" and by "!" patterns. We use a strict let-binding form to give the same effect. Note that our language is not call-by-need like Haskell, in that we do not have a stateful, sharing-based operational semantics. However, since the only unencapsulated effects in our language and Haskell are an uncatchable error and divergence, this difference is not observable since any effect would end evaluation.

We review the common syntax in Figure 6.1

1 The only exception, as mentioned in Chapter 5, are the rules regarding disjointness of type constructors.

$$
\begin{array}{lrcl}
\text{Types} & A & ::= & ? \mid A \to A \mid A \times A \mid \texttt{Bool} \\
\text{Ground types} & G & ::= & ? \to ? \mid ? \times ? \mid \texttt{Bool} \\
\text{Terms} & M, N & ::= & \mathcal{U} \mid x \mid \texttt{let } x = M; N \mid \langle A \Leftarrow A \rangle M \\
 & & \mid & (M, N) \mid \pi_i M \\
 & & \mid & \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \mid \lambda x : A.M \mid M\ N
\end{array}
$$

Figure 6.1: Cast Calculus Syntax

6.1 CALL-BY-VALUE

First, we review call-by-value evaluation order in Figure 6.2. We first define the syntax of values and evaluation contexts. Values include the ordinary STLC values, and additionally tagged values of the dynamic type $\langle ? \Leftarrow G \rangle V$. We fix the evaluation order by defining *evaluation contexts* which we write as E since they correspond to CBPV stacks.

Next, we present the operational semantics of our calculus. The first six rules correspond to ordinary CBV reductions so we don't bother to name them. The remaining rules are specific to casts. First, ?Id says that casting from ? to ? is the identity. The next two rules DecompUp,DecompDn break down complex casts to and from the dynamic type to go through the associated ground type (note that for any type A, if $A \neq ?$ then there is precisely one ground type G such that $A \sqsubset G$). The next two rules TagMatch,TagMismatch say that casting a tagged value $\langle ? \Leftarrow G \rangle V$ to a ground type G' succeeds if the tag is the same ($G = G'$) and fails if the tag is different ($G \neq G'$). Finally, the Silly rule is a catch all that says when casting between two completely unrelated types, the cast fails. The remaining rules give the behavior of casts between two types with the same head connective, implementing the wrapping strategy.

6.1.1 *From CBV to GTT*

Our goal is to show that the operational reductions of our CBV cast calculus are in a sense *derivable* from the axioms of GTT. To make this concrete, we will define a type-preserving translation of our CBV calculus terms M into GTT computations M^c and prove that for almost every reduction $M \mapsto N$ in the CBV calculus, $M^c \sqsupseteq\sqsubseteq N^c$ is provable in GTT. The only rules that do not follow from the axioms of GTT are those that result in errors: TagMismatch and Silly. The reason for this is that nothing in GTT encodes the "disjointness" of different type connectives, and we consider this part of the language to be a true design decision. We explore in Chapter 7 some alternative design choices for gradual languages.

$$
\begin{array}{lrcl}
\text{Values} & V & ::= & \langle ? \Leftarrow G \rangle V \mid \lambda x : A.M \mid (V, V) \mid \texttt{true} \mid \texttt{false} \\
\text{Evaluation Contexts} & E & ::= & \bullet \mid \langle B \Leftarrow A \rangle E \mid \texttt{let } x = E; N \mid (E, N) \mid (V, E) \\
& & \mid & \pi_i E \\
& & \mid & \texttt{if } E \texttt{ then } N_1 \texttt{ else } N_2 \mid E\, N \mid V\, E \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Substitutions} & \gamma & ::= & \cdot \mid \gamma, V/x
\end{array}
$$

$$
\begin{array}{ll}
E[\texttt{let } x = V; N] & \mapsto^v E[N[V/x]] \\
E[(\lambda x : A.M)\, V] & \mapsto^v E[M[V/x]] \\
E[\pi_i(V_1, V_2)] & \mapsto^v E[V_i] \\
E[\texttt{if true then } N_1 \texttt{ else } N_2] & \mapsto^v E[N_1] \\
E[\texttt{if false then } N_1 \texttt{ else } N_2] & \mapsto^v E[N_2]
\end{array}
$$

?ID
$$E[\langle ? \Leftarrow ? \rangle V] \mapsto^v E[V]$$

DECOMPUP
$$\frac{A \sqsubseteq G \qquad A \neq G}{E[\langle ? \Leftarrow A \rangle V] \mapsto^v E[\langle ? \Leftarrow G \rangle \langle G \Leftarrow A \rangle V]}$$

DECOMPDN
$$\frac{A \sqsubseteq G \qquad A \neq G}{E[\langle A \Leftarrow ? \rangle V] \mapsto^v E[\langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle V]}$$

TAGMATCH
$$E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle V] \mapsto^v E[V]$$

TAGMISMATCH
$$\frac{G \neq G'}{E[\langle G' \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle V] \mapsto^v \mathcal{U}}$$

EILLY
$$\frac{A \sqsubseteq G_A \qquad B \sqsubseteq G_B \qquad G_A \neq G_B}{E[\langle B \Leftarrow A \rangle V] \mapsto^v \mathcal{U}}$$

$\to$CAST
$$E[\langle A_1' \to A_2' \Leftarrow A_1 \to A_2 \rangle V] \mapsto^v E[\lambda x : A_1'.\langle A_2' \Leftarrow A_2 \rangle (V\, (\langle A_1 \Leftarrow A_1' \rangle x))]$$

$$E[\langle A_1' \times A_2' \Leftarrow A_1 \times A_2 \rangle (V_1, V_2)] \mapsto^v E[(\langle A_1' \Leftarrow A_1 \rangle V_1, \langle A_2' \Leftarrow A_2 \rangle V_2)]$$

Figure 6.2: CBV Cast Calculus Operational Semantics

If $\Gamma \vdash M : A$ then $\Gamma^{ty} \vdash M^c : \underline{F} A^{ty}$

$$
\begin{aligned}
?^{ty} &= ? \\
(A \to A')^{ty} &= U(A^{ty} \to \underline{F} A'^{ty}) \\
(A_1 \times A_2)^{ty} &= A_1^{ty} \times A_2^{ty} \\
\texttt{Bool}^{ty} &= 1 + 1
\end{aligned}
$$

$$
\begin{aligned}
x^c &= \texttt{ret}\ x \\
(\texttt{let}\ x = M; N)^c &= x \leftarrow M^c; N^c \\
(\langle A_1 \Leftarrow A_2 \rangle M)^c &= \langle F A_2^{ty} \swarrow F? \rangle \langle\langle F? \nwarrow F A_1^{ty} \rangle\rangle [M^c] \\
(\lambda x : A.M)^c &= \texttt{ret}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c)) \\
(M\,N)^c &= f \leftarrow M^c; x \leftarrow N^c; \texttt{force}\ f\ x \\
(M_1, M_2)^c &= x_1 \leftarrow M_1^c; x_2 \leftarrow M_2^c; \texttt{ret}\ (x_1, x_2) \\
(\pi_i M)^c &= z \leftarrow M^c; \texttt{let}\ (x_1, x_2) = z; \texttt{ret}\ x_i \\
\texttt{true}^c &= \texttt{ret}\ \texttt{inl}\ () \\
\texttt{false}^c &= \texttt{ret}\ \texttt{inr}\ () \\
(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2)^c &= z \leftarrow M^c; \texttt{case}\ z \{N_1^c \mid N_2^c\}
\end{aligned}
\tag{6.1}
$$

Figure 6.3: CBV to GTT translation

We define the type and term translation in Figure 6.3. First, we translate CBV types A to CBPV value types $\underline{F}A$, with the only non-trivial case being the translation of function types. Next the computation type translation is mostly straightforward, making the evaluation order explicit using $x \leftarrow M; N$. The only non-standard case is the rule for casts, where as discussed in Chapter 3, we define the semantics of all casts to factorize as an upcast to the dynamic type followed by a downcast out of the dynamic type. Note finally that since we are working in CBV, we don't ever need to use the computation dynamic type $\underline{?}$.

The rest of this section proves the following theorem:

Theorem 94. *If $M \mapsto^v N$ by any rule except* TagMismatch *or* Silly, *then $M^c \sqsupseteq\sqsubseteq N^c$.*

To reason about substitution and plugging in evaluation contexts in the correctness proofs, we additionally define a *value* translation that directly translates CBV values to GTT values and a *stack* translation that directly translates CBV evaluation contexts to GTT stacks in Figure 6.4

We then prove a few correctness principles for these with respect to the term translation.

Lemma 95. $V^c \sqsupseteq\sqsubseteq \texttt{ret}\ V^v$

$$
\begin{aligned}
(\langle G \Leftarrow ? \rangle V)^v &= \langle ? \leftharpoonup G^{ty} \rangle V \\
(\lambda x : A.M)^v &= \mathtt{thunk}\ (\lambda x : A^{ty}.M^c) \\
(V_1, V_2)^v &= (V_1^v, V_2^v) \\
\mathtt{true}^v &= \mathtt{inl}\ () \\
\mathtt{false}^v &= \mathtt{inr}\ ()
\end{aligned}
$$

$$
\begin{aligned}
\bullet^s &= \bullet \\
(\mathtt{let}\ x = E; N)^s &= x \leftarrow E^s; N^c \\
(\langle A_1 \Leftarrow A_2 \rangle E)^s &= x \leftarrow E^s; \langle FA_2^{ty} \swarrow F? \rangle (\mathtt{ret}\ \langle ? \leftharpoonup A_1^{ty} \rangle x) \\
(E\ N)^s &= f \leftarrow E^s; x \leftarrow N^c; \mathtt{force}\ f\ x \\
(V\ E)^s &= x \leftarrow E^s; \mathtt{force}\ V^v\ x \\
(E_1, M_2)^s &= x_1 \leftarrow E_1^c; x_2 \leftarrow M_2^c; \mathtt{ret}\ (x_1, x_2) \\
(V_1, E_2)^s &= x_2 \leftarrow E_2^c; \mathtt{ret}\ (V_1^v, x_2) \\
(\pi_i E)^s &= z \leftarrow E^s; \mathtt{let}\ (x_1, x_2) = z; \mathtt{ret}\ x_i \\
(\mathtt{if}\ E\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2)^s &= z \leftarrow E^s; \mathtt{case}\ z \{x_1.N_1^c \mid x_2.N_2^c\}
\end{aligned}
\tag{6.2}
$$

Figure 6.4: CBV Value and Stack translations

Proof. By induction on V.

- $\langle ? \Leftarrow G \rangle V$:

$$
\begin{aligned}
(\langle ? \Leftarrow G \rangle V)^c &= \langle F? \swarrow F? \rangle \langle\!\langle \underline{F}? \leftharpoonup \underline{F} G^{ty} \rangle\!\rangle V^c && \text{(defn.)} \\
&\sqsupseteq\sqsubseteq \langle\!\langle \underline{F}? \leftharpoonup \underline{F} G^{ty} \rangle\!\rangle V^c && \text{(Theorem 72)} \\
&= x \leftarrow V^c; \mathtt{ret}\ \langle ? \leftharpoonup G^{ty} \rangle x && \text{(defn.)} \\
&\sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ V^v; \mathtt{ret}\ \langle ? \leftharpoonup G^{ty} \rangle x && \text{(defn.)} \\
&\sqsupseteq\sqsubseteq \mathtt{ret}\ \langle ? \leftharpoonup G^{ty} \rangle V^v && (F\beta) \\
&= \mathtt{ret}\ (\langle ? \Leftarrow G \rangle V)^v && \text{(defn.)}
\end{aligned}
$$

- $\lambda x : A.M$: immediate by reflexivity.
- $()$: immediate by reflexivity.
- (V_1, V_2):

$$
\begin{aligned}
(V_1, V_2)^c &= x_1 \leftarrow V_1^c; x_2 \leftarrow V_2^c; \mathtt{ret}\ (x_1, x_2) && \text{(definition)} \\
&\sqsupseteq\sqsubseteq x_1 \leftarrow \mathtt{ret}\ V_1^v; x_2 \leftarrow \mathtt{ret}\ V_2^v; \mathtt{ret}\ (x_1, x_2) && \text{(I.H., twice)} \\
&\sqsupseteq\sqsubseteq \mathtt{ret}\ (V_1, V_2) && (F\beta \text{ twice})
\end{aligned}
$$

- $\texttt{inl}\ V$:

$$\begin{aligned}
(\texttt{inl}\ V)^c &= x \leftarrow V^c; \texttt{ret inl}\ x && \text{(definition)}\\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{ret}\ V^v; \texttt{ret inl}\ x && \text{(I.H.)}\\
&\sqsupseteq\sqsubseteq \texttt{ret inl}\ V^v && (F\beta)
\end{aligned}$$

- $\texttt{inr}\ V$: similar to $\texttt{inl}$ case.

□

Lemma 96. $(M[V/x])^c \sqsupseteq\sqsubseteq M^c[V^v/x]$

Proof. By induction on M. All cases but variable are by congruence and inductive hypothesis.

- $M = x$:

$$\begin{aligned}
(x[V/x])^c &= V^c && \text{(def. substitution)}\\
&\sqsupseteq\sqsubseteq \texttt{ret}\ V^v && \text{(Lemma 95)}\\
&= (\texttt{ret}\ x)[V^v/x] && \text{(def. substitution)}\\
&= (x^c)[V^v/x] && \text{(def. substitution)}
\end{aligned}$$

- $M = y \neq x$:

$$\begin{aligned}
(y[V/x])^c &= y^c && \text{(def. subst.)}\\
&\sqsupseteq\sqsubseteq \texttt{ret}\ y && \text{(def.)}\\
&\sqsupseteq\sqsubseteq (\texttt{ret}\ y)[V/x] && \text{(def. subst.)}
\end{aligned}$$

□

Lemma 97. $(E[M])^c \sqsupseteq\sqsubseteq E^s[M^c]$

Proof. By induction on E. Most cases are straightforward by congruence and induction hypothesis. We show the other cases.

- $E = V\,E$:

$$\begin{aligned}
((V\,E)[M])^c &= (V\,(E[M]))^c && \text{(defn. plugging)}\\
&= f \leftarrow V^c; x \leftarrow (E[M])^c; \texttt{force}\ f\ x && \text{(defn.)}\\
&\sqsupseteq\sqsubseteq f \leftarrow \texttt{ret}\ V^v; x \leftarrow (E[M])^c; \texttt{force}\ f\ x && \text{(Lemma 95)}\\
&\sqsupseteq\sqsubseteq x \leftarrow (E[M])^c; \texttt{force}\ V^v\ x && (F\beta)\\
&\sqsupseteq\sqsubseteq x \leftarrow E^s[M^c]; \texttt{force}\ V^v\ x && \text{(I.H.)}\\
&\sqsupseteq\sqsubseteq (x \leftarrow E^s; \texttt{force}\ V^v\ x)[M^c] && \text{(defn. of plug)}\\
&= (V\,E)^s[M^c]
\end{aligned}$$

- $E = (V_1, E_2)$

$$\begin{aligned}
((V_1, E_2)[M])^c &= (V_1, E_2[M])^c && \text{(defn. plugging)}\\
&= x_1 \leftarrow V_1^c; x_2 \leftarrow E_2[M]^c; \mathtt{ret}\ (x_1, x_2) && \text{(defn.)}\\
&\sqsupseteq\sqsubseteq x_1 \leftarrow \mathtt{ret}\ V_1^v; x_2 \leftarrow E_2[M]^c; \mathtt{ret}\ (x_1, x_2) && \text{(Lemma 95)}\\
&\sqsupseteq\sqsubseteq x_2 \leftarrow E_2[M]^c; \mathtt{ret}\ (V_1^v, x_2) && (F\beta)\\
&\sqsupseteq\sqsubseteq x_2 \leftarrow E_2^s[M^c]; \mathtt{ret}\ (V_1^v, x_2) && \text{(I.H.)}\\
&= (x_2 \leftarrow E_2^s; \mathtt{ret}\ (V_1^v, x_2))[M^c] && \text{(defn. plugging)}\\
&= (V_1, E_2^s)[M^c] && \text{(defn.)}
\end{aligned}$$

□

We will also need the following simple GTT lemma to help reason about casts. We prove a version for value types and computation types that will be used in the CBN section.

Lemma 98 (Any Middle Type will Do). *For any value types $A_1, A_2 \sqsubseteq A'$, $\langle FA_2 \swarrow F?\rangle\langle\!\langle F? \nwarrow FA_1\rangle\!\rangle M \sqsupseteq\sqsubseteq \langle FA_2 \swarrow FA'\rangle\langle\!\langle FA' \nwarrow FA_1\rangle\!\rangle M$*

Similarly, for any computation types $B_1, B_2 \sqsubseteq B'$, $\langle\!\langle UB_2 \swarrow U\underline{¿}\rangle\!\rangle\langle U\underline{¿} \nwarrow UB_1\rangle V \sqsupseteq\sqsubseteq \langle\!\langle UB_2 \swarrow UB'\rangle\!\rangle\langle UB' \nwarrow UB_1\rangle VM$

Proof.

$$\begin{aligned}
&\langle FA_2 \swarrow F?\rangle\langle\!\langle F? \nwarrow FA_1\rangle\!\rangle M\\
&\sqsupseteq\sqsubseteq \langle FA_2 \swarrow FA'\rangle\langle FA' \swarrow F?\rangle\langle\!\langle F? \nwarrow FA'\rangle\!\rangle\langle\!\langle FA' \nwarrow FA_1\rangle\!\rangle M && \text{(Theorem 72)}\\
&\sqsupseteq\sqsubseteq \langle FA_2 \swarrow FA'\rangle\langle\!\langle FA' \nwarrow FA_1\rangle\!\rangle M && \text{(retraction)}
\end{aligned}$$

□

And now we can prove the CBV correctness theorem.

PROOF OF THEOREM 94

Proof. In all cases, by Lemma 97, congruence and $E[\mathcal{V}] \sqsupseteq\sqsubseteq \mathcal{V}$, it is sufficient to consider the case that $E = \bullet$.

First, we have the cases not involving casts, which are standard for the embedding of call-by-value into call-by-push-value.

- $\mathtt{let}\ x = V; N \mapsto^v N[V/x]$

$$\begin{aligned}
(\mathtt{let}\ x = V; N)^c &= x \leftarrow V^c; N^c\\
&\sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ V^v; N^c\\
&\sqsupseteq\sqsubseteq N^c[V^v/x]\\
&\sqsupseteq\sqsubseteq (N[V/x]^c)
\end{aligned} \tag{6.3}$$

- $(\lambda x : A.M)\, V \mapsto^v M[V/x]$

$$\begin{aligned}((\lambda x : A.M)\, V)^c &= f \leftarrow (\texttt{ret}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c)));x \leftarrow V^c;\texttt{force}\ f\, x \\ &\sqsupseteq\sqsubseteq x \leftarrow V^c;\texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\, x \\ &\sqsupseteq\sqsubseteq x \leftarrow \texttt{ret}\ V^v;\texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\, x \\ &\sqsupseteq\sqsubseteq \texttt{force}\ (\texttt{thunk}\ (\lambda x : A^{ty}.M^c))\, V^v \\ &\sqsupseteq\sqsubseteq (\lambda x : A^{ty}.M^c)\, V^v \\ &\sqsupseteq\sqsubseteq M^c[V^v/x] \\ &\sqsupseteq\sqsubseteq (M[V/x])^c\end{aligned} \tag{6.4}$$

- $\pi_i(V_1, V_2) \mapsto^v V_i$

$$\begin{aligned}\pi_i(V_1, V_2)^c &= z \leftarrow (V_1, V_2)^c;\texttt{let}\ (x_1, x_2) = z;\texttt{ret}\ x_i \\ &\sqsupseteq\sqsubseteq z \leftarrow \texttt{ret}\ (V_1^v, V_2^v);\texttt{let}\ (x_1, x_2) = z; N^c \\ &\sqsupseteq\sqsubseteq \texttt{let}\ (x_1, x_2) = (V_1^v, V_2^v);\texttt{ret}\ x_i \\ &\sqsupseteq\sqsubseteq \texttt{ret}\ x_i[V_1^v/x_1][V_2^v/x_2] \\ &\sqsupseteq\sqsubseteq \texttt{ret}\ V_i\end{aligned} \tag{6.5}$$

- $\texttt{if true then}\ N_1\ \texttt{else}\ N_2 \mapsto^v N_1$

$$\begin{aligned}(\texttt{if true then}\ N_1\ \texttt{else}\ N_2)^c &= z \leftarrow \texttt{ret inl}\ ();\texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\} \\ &\sqsupseteq\sqsubseteq \texttt{case inl}\ ()\{x_1.N_1^c \mid x_2.N_2^c\} \\ &\sqsupseteq\sqsubseteq N_1^c\end{aligned} \tag{6.6}$$

- $\texttt{if false then}\ N_1\ \texttt{else}\ N_2 \mapsto^v N_2$, similar to previous.

Next, we have the interesting cases, those specific to gradual type casts/GTT.

- $\langle ? \Leftarrow ? \rangle V \mapsto^v V$:

$$\begin{aligned}(\langle ? \Leftarrow ? \rangle V)^c &= \langle F? \swarrow F? \rangle \langle\!\langle F? \nwarrow F? \rangle\!\rangle [V^c] \\ &\sqsupseteq\sqsubseteq V^c \qquad \text{(Theorem 72)}\end{aligned}$$

- $\langle ? \Leftarrow A \rangle V \mapsto^v \langle ? \Leftarrow G \rangle \langle G \Leftarrow A \rangle V$ where $A \sqsubseteq G$

$$\begin{aligned}&\langle ? \Leftarrow A \rangle V^c \\ &= \langle F? \swarrow F? \rangle \langle\!\langle F? \nwarrow FA^{ty} \rangle\!\rangle [V^c] \\ &\sqsupseteq\sqsubseteq \langle F? \swarrow F? \rangle \langle\!\langle F? \nwarrow FG^{ty} \rangle\!\rangle \langle\!\langle FG^{ty} \nwarrow FA^{ty} \rangle\!\rangle [V^c] \\ &\sqsupseteq\sqsubseteq \langle F? \swarrow F? \rangle \langle\!\langle F? \nwarrow FG^{ty} \rangle\!\rangle \langle FG^{ty} \swarrow F? \rangle \langle\!\langle F? \nwarrow FG^{ty} \rangle\!\rangle \langle\!\langle FG^{ty} \nwarrow FA^{ty} \rangle\!\rangle [V^c] \\ &\sqsupseteq\sqsubseteq \langle F? \swarrow F? \rangle \langle\!\langle F? \nwarrow FG^{ty} \rangle\!\rangle \langle FG^{ty} \swarrow F? \rangle \langle\!\langle F? \nwarrow FA^{ty} \rangle\!\rangle [V^c] \\ &= (\langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle V)^c\end{aligned} \tag{6.7}$$

- $\langle A \Leftarrow ?\rangle V \mapsto^v \langle A \Leftarrow G\rangle\langle G \Leftarrow ?\rangle V$: similar to previous case.
- $\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V \mapsto^v V$

$$\begin{aligned}(\langle G \Leftarrow ?\rangle\langle ? \Leftarrow G\rangle V)^c &= \langle FG^{ty} \swarrow F?\rangle\langle\!\langle F? \nwarrow F?\rangle\!\rangle\langle F? \swarrow F?\rangle\langle\!\langle F? \nwarrow FG^{ty}\rangle\!\rangle[V^c] \\ &\sqsupseteq\sqsubseteq \langle FG^{ty} \swarrow F?\rangle\langle\!\langle F? \nwarrow FG^{ty}\rangle\!\rangle[V^c] && \text{(Theorem 72)} \\ &\sqsupseteq\sqsubseteq V^c && \text{(retraction)}\end{aligned}$$

- $\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle V \mapsto^v \lambda x : A_1'.\langle A_2' \Leftarrow A_2\rangle(V\,(\langle A_1 \Leftarrow A_1'\rangle x))$

$$\begin{aligned}&(\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle V)^c \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow F?\rangle\langle\!\langle F? \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle V^c \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle\langle\!\langle FU(? \to F?) \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle V^c \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle\langle\!\langle FU(? \to F?) \nwarrow FU(A_1^{ty} \to FA_2^{ty})\rangle\!\rangle[\mathtt{ret}\ V^v] \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \mathtt{ret}\ \langle U(? \to F?) \nwarrow U(A_1^{ty} \to FA_2^{ty})\rangle V^v \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \\ &\qquad \mathtt{ret}\ \mathtt{thunk} \begin{pmatrix} \lambda x'.x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\mathtt{ret}\ x'); \\ \mathtt{force}\ (\langle U\underline{F}? \nwarrow U\underline{F}A_2^{ty}\rangle(\mathtt{thunk}\ (\mathtt{force}\ V^v\ x))) \end{pmatrix} \\ &\sqsupseteq\sqsubseteq \langle FU(A_1'^{ty} \to FA_2'^{ty}) \swarrow FU(? \to F?)\rangle \\ &\qquad \mathtt{ret}\ \mathtt{thunk}\ (\lambda x'.x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\mathtt{ret}\ x'); \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x)) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\langle A_1'^{ty} \to FA_2'^{ty} \swarrow ? \to F?\rangle(\lambda x'.x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\mathtt{ret}\ x'); \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x))) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle((\lambda x'.x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\mathtt{ret}\ x'); \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x))) \\ &\qquad\qquad (\langle ? \nwarrow A_1'^{ty}\rangle y')) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle((x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle(\mathtt{ret}\ (\langle ? \nwarrow A_1'^{ty}\rangle y'); \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x)) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle((x \leftarrow \langle \underline{F}A_1^{ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1'^{ty}\rangle\!\rangle \mathtt{ret}\ y'; \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle((x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^{ty}; \langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(\mathtt{force}\ V^v\ x)))) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_2^{ty}\rangle\!\rangle(x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^{ty}; (\mathtt{force}\ V^v\ x))) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{ty}\rangle\!\rangle(f \leftarrow V^c; x \leftarrow (\langle A_1 \Leftarrow A_1'\rangle y')^c; (\mathtt{force}\ f\ x))) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.\langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}A_1^{ty}\rangle\!\rangle(V\,(\langle A_1 \Leftarrow A_1'\rangle y'))^c) \\ &\sqsupseteq\sqsubseteq \mathtt{ret}\ \mathtt{thunk}\ (\lambda y'.(\langle A_2' \Leftarrow A_2\rangle(V\,(\langle A_1 \Leftarrow A_1'\rangle y')))^c) \\ &\sqsupseteq\sqsubseteq (\lambda y'.\langle A_2' \Leftarrow A_2\rangle(V\,(\langle A_1 \Leftarrow A_1'\rangle y')))^c \end{aligned} \tag{6.8}$$

- $\langle A_1' \times A_2' \Leftarrow A_1 \times A_2\rangle(V_1, V_2) \mapsto^v (\langle A_1' \Leftarrow A_1\rangle V_1, \langle A_2' \Leftarrow A_2\rangle V_2)$

$$
\begin{aligned}
&(\langle A_1' \times A_2' \Leftarrow A_1 \times A_2\rangle(V_1, V_2))^c \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}?\rangle\langle\!\langle \underline{F}? \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle(V_1, V_2)^c \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}(? \times ?)\rangle\langle\!\langle \underline{F}(? \times ?) \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle(V_1, V_2)^c \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}(? \times ?)\rangle\langle\!\langle \underline{F}(? \times ?) \nwarrow \underline{F}(A_1^{ty} \times A_2^{ty})\rangle\!\rangle \mathtt{ret}\ (V_1^v, V_2^v) \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}(? \times ?)\rangle(\mathtt{ret}\ \langle (? \times ?) \nwarrow (A_1^{ty} \times A_2^{ty})\rangle(V_1^v, V_2^v)) \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}(? \times ?)\rangle \\
&\qquad (\mathtt{ret}\ (\mathtt{let}\ (x_1, x_2) = (V_1^v, V_2^v); (\langle ? \nwarrow A_1^{ty}\rangle x_1, \langle ? \nwarrow A_2^{ty}\rangle x_2))) \\
&\sqsupseteq\sqsubseteq \langle \underline{F}(A_1'^{ty} \times A_2'^{ty}) \swarrow \underline{F}(? \times ?)\rangle(\mathtt{ret}\ (\langle ? \nwarrow A_1^{ty}\rangle V_1^v, \langle ? \nwarrow A_2^{ty}\rangle x_2)) \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (y_1, y_2) = (\langle ? \nwarrow A_1^{ty}\rangle V_1^v, \langle ? \nwarrow A_2^{ty}\rangle V_2^v); \\
&\qquad x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \mathtt{ret}\ y_1; \\
&\qquad x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \mathtt{ret}\ y_2; \mathtt{ret}\ (x_1', x_2') \\
&\sqsupseteq\sqsubseteq x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle \mathtt{ret}\ \langle ? \nwarrow A_1^{ty}\rangle V_1^v; \\
&\qquad x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle \mathtt{ret}\ \langle ? \nwarrow A_2^{ty}\rangle V_2^v; \mathtt{ret}\ (x_1', x_2') \\
&\sqsupseteq\sqsubseteq x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle ? \nwarrow A_1^{ty}\rangle\!\rangle \mathtt{ret}\ V_1^v; \\
&\qquad x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle ? \nwarrow A_2^{ty}\rangle\!\rangle \mathtt{ret}\ V_2^v; \mathtt{ret}\ (x_1', x_2') \\
&\sqsupseteq\sqsubseteq x_1' \leftarrow \langle \underline{F}A_1'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle ? \nwarrow A_1^{ty}\rangle\!\rangle V_1^c; \\
&\qquad x_2' \leftarrow \langle \underline{F}A_2'^{ty} \swarrow \underline{F}?\rangle\langle\!\langle ? \nwarrow A_2^{ty}\rangle\!\rangle V_2^c; \mathtt{ret}\ (x_1', x_2') \\
&\sqsupseteq\sqsubseteq x_1' \leftarrow \langle A_1' \Leftarrow A_1\rangle V_1^c; x_2' \leftarrow \langle A_2' \Leftarrow A_2\rangle V_2^c; \mathtt{ret}\ (x_1', x_2') \\
&= (\langle A_1' \Leftarrow A_1\rangle V_1, \langle A_2' \Leftarrow A_2\rangle V_2)^c
\end{aligned}
\tag{6.9}
$$

□

6.2 CALL-BY-NAME

Next, we cover call-by-name evaluation order in Figure 6.5. First, we define evaluation contexts, which correspond to CBPV stacks, and find the redex within the larger term. Unlike CBV, there's no non-trivial notion of value. Then we have the operational rules. Again we start with the β reduction rules. Let-binding and function application are both lazy, substituting an unreduced term for a variable. Similarly, projection doesn't reduce the side of the pair that is not being projected. Finally, if-statements are essentially the same as in CBV since booleans are first-order data. Then the cast rules are all analogous to CBV but are non-strict in that they do not reduce the scrutinee to a value first. In particular the function and product rules immediately step to the wrapping definition.

$$E \quad ::= [\cdot] \mid E\ M \mid \pi_i E \mid \texttt{if}\ E\ \texttt{then}\ M_t\ \texttt{else}\ M_f \mid \langle G \Leftarrow ? \rangle E$$

$$\begin{array}{ll}
E[\texttt{let}\ x = M; N] & \mapsto^n E[N[M/x]] \\
E[(\lambda x.M)N] & \mapsto^n E[M[N/x]] \\
E[\pi_i\ (M_1, M_2)] & \mapsto^n E[M_i] \\
E[\texttt{if true then}\ M_t\ \texttt{else}\ M_f] & \mapsto^n E[M_t] \\
E[\texttt{if false then}\ M_t\ \texttt{else}\ M_f] & \mapsto^n E[M_f]
\end{array}$$

?ID
$$E[\langle ? \Leftarrow ? \rangle M] \mapsto^n E[M]$$

DECOMPUP
$$\frac{A \sqsubseteq G}{E[\langle ? \Leftarrow A \rangle M] \mapsto^n E[\langle ? \Leftarrow G \rangle \langle G \Leftarrow A \rangle M]}$$

DECOMPDN
$$\frac{A \sqsubseteq G}{E[\langle A \Leftarrow ? \rangle M] \mapsto^n E[\langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle M]}$$

TAGMATCH
$$E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle M] \mapsto^n E[M]$$

TAGMISMATCH
$$\frac{G \neq G'}{E[\langle G' \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle M] \mapsto^n \mathcal{U}}$$

SILLY
$$\frac{A \sqsubseteq G_A \qquad B \sqsubseteq G_B \qquad G_A \neq G_B}{E[\langle B \Leftarrow A \rangle M] \mapsto^n \mathcal{U}}$$

$$\begin{array}{ll}
E[(\langle A \to B \Leftarrow A' \to B' \rangle M)] & \mapsto^n E[\lambda x.\langle B \Leftarrow B' \rangle (M\ (\langle A' \Leftarrow A \rangle x))] \\
E[(\langle A_1 \times A_2 \Leftarrow A_1' \times A_2' \rangle M)] & \mapsto^n E[(\langle A_1 \Leftarrow A_1' \rangle (\pi_1 M), \langle A_2 \Leftarrow A_2' \rangle (\pi_2 M))]
\end{array}$$

Figure 6.5: Call-by-name Reduction

If $\Gamma \vdash M : A$, then $U\Gamma^{ty} \vdash M^c : A^{ty}$

$$
\begin{aligned}
?^{ty} &= \underline{¿} \\
(A \to A')^{ty} &= UA^{ty} \to A'^{ty} \\
(A_1 \times A_2)^{ty} &= A_1^{ty} \,\&\, A_2^{ty} \\
\texttt{Bool}^{ty} &= \underline{F}(1+1)
\end{aligned}
$$

$$
\begin{aligned}
x^c &= \texttt{force}\ x \\
(\texttt{let}\ x = M; N)^c &= \texttt{let}\ x = \texttt{thunk}\ M^c; N^c \\
(\langle A_2 \Leftarrow A_1 \rangle M)^c &= \texttt{force}\ \langle\langle UA_2^{ty} \swarrow U\underline{¿} \rangle\rangle \langle U\underline{¿} \nwarrow UA_1^{ty} \rangle (\texttt{thunk}\ M^c) \\
(\lambda x : A.M)^c &= (\lambda x : UA^{ty}.M^c) \\
M\ N^c &= M^c\ (\texttt{thunk}\ N^c) \\
(M_1, M_2)^c &= \{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\} \\
\pi_i M^c &= \pi_i M^c \\
\texttt{true}^c &= \texttt{ret inl}\ () \\
\texttt{false}^c &= \texttt{ret inr}\ () \\
(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2)^c &= z \leftarrow M^c; \texttt{case}\ z\{x_1.\ N_1^c \mid x_2.\ N_2^c\}
\end{aligned}
\tag{6.10}
$$

Figure 6.6: CBN to GTT translation

Next, we define the elaboration of CBN into GTT in Figure 6.6. This follows Levy's original CBN translation. First, CBN types are interpreted as GTT *computation* types, which matches the fact that CBN evaluation order does not have a notion of value. In particular, the dynamic type is interpreted as the computation dynamic type $\underline{¿}$. Next, variables all denote *thunks* and a usage of a variable forces it. In other ways the translation is somewhat simpler than CBV as most forms translate directly to the corresponding form of CBPV but with the insertion of some thunks.

Next we prove some analogous lemmas before proving translation correctness.

To reason about plugging in evaluation contexts in the correctness proofs, we additionally define a *stack* translation that directly translates CBN evaluation contexts to GTT stacks in Figure 6.4

We then prove a few correctness principles for these with respect to the term translation.

Lemma 99. $(M[N/x])^c \sqsupseteq\sqsubseteq M^c[\texttt{thunk}\ (N^c)/x]$

Proof. By induction on M. All cases but variable are by congruence and inductive hypothesis.

$$
\begin{aligned}
\bullet^s &= \bullet \\
(\langle G \Leftarrow ? \rangle E)^s &= \langle G^{ty} \swarrow \underline{¿} \rangle E^s \\
(E\, N)^s &= E^s\ (\texttt{thunk}\ N^c) \\
(\pi_i E)^s &= \pi_i E^s \\
(\texttt{if}\ E\ \texttt{then}\ N_1\ \texttt{else}\ N_2)^s &= z \leftarrow E^s; \texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\}
\end{aligned}
\tag{6.11}
$$

Figure 6.7: CBN Value and Stack translations

- $M = x$:

$$
\begin{aligned}
(x[N/x])^c &= N^c && \text{(def. substitution)} \\
&\sqsupseteq\sqsubseteq \texttt{force}\ (\texttt{thunk}\ N^v) && (U\beta) \\
&= (\texttt{force}\ x)[\texttt{thunk}\ N^c/x] && \text{(def. substitution)} \\
&= (x^c)[\texttt{thunk}\ N^c/x] && \text{(def. trans)}
\end{aligned}
$$

- $M = y \neq x$:

$$
\begin{aligned}
(y[N/x])^c &= \texttt{force}\ y^c && \text{(def. trans.)} \\
&\sqsupseteq\sqsubseteq (\texttt{force}\ y)[\texttt{thunk}\ N^c/x] && \text{(def. subst.)}
\end{aligned}
$$

□

Lemma 100. $(E[M])^c \sqsupseteq\sqsubseteq E^s[M^c]$

Proof. By induction on E. Most cases are straightforward by congruence and induction hypothesis. The only other case is the cast case:

- $E = \langle G \Leftarrow ? \rangle E$:

$$
\begin{aligned}
&(\langle G \Leftarrow ? \rangle E)[M]^c \\
&= (\langle G \Leftarrow ? \rangle E[M])^c && \text{(defn. plugging)} \\
&= \texttt{force}\ \langle\!\langle UG^{ty} \swarrow U\underline{¿} \rangle\!\rangle \langle U\underline{¿} \nwarrow U\underline{¿} \rangle \texttt{thunk}\ E[M]^c && \text{(defn.)} \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UG^{ty} \swarrow U\underline{¿} \rangle\!\rangle \texttt{thunk}\ E[M]^c && \text{(decomposition)} \\
&= \texttt{force}\ (\texttt{thunk}\ (\langle G^{ty} \swarrow \underline{¿} \rangle (\texttt{force}\ \texttt{thunk}\ E[M]^c))) \\
& && \text{(defn.)} \\
&\sqsupseteq\sqsubseteq \langle G^{ty} \swarrow \underline{¿} \rangle E[M]^c && \text{(U}\beta\text{, twice)} \\
&\sqsupseteq\sqsubseteq \langle G^{ty} \swarrow \underline{¿} \rangle E^s[M^c] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq (\langle G^{ty} \swarrow \underline{¿} \rangle E^s)[M^c] && \text{(def)} \\
&\sqsupseteq\sqsubseteq (\langle G \Leftarrow ? \rangle E^s)[M^c] && \text{(def)}
\end{aligned}
$$

□

And we can now establish our central theorem.

Theorem 101. *If $M \mapsto^n N$ by any rule except* TagMismatch *or* Silly, *then $M^c \sqsupseteq\sqsubseteq N^c$.*

Proof. In all cases, by Lemma 100, congruence and $E[\mho] \sqsupseteq\sqsubseteq \mho$, it is sufficient to consider the case that $E = \bullet$.

First, we have the cases not involving casts, which are standard for the embedding of call-by-value into call-by-push-value.

- $\texttt{let}\ x = M; N \mapsto^n N[M/x]$

$$\begin{aligned}(\texttt{let}\ x = M; N)^c &= \texttt{let}\ x = \texttt{thunk}\ M^c; N^c \\ &\sqsupseteq\sqsubseteq N^c[\texttt{thunk}\ M^c] \\ &\sqsupseteq\sqsubseteq (N[M/x]^c)\end{aligned} \tag{6.12}$$

- $(\lambda x : A.M)\ N \mapsto^n M[N/x]$

$$\begin{aligned}((\lambda x : A.M)\ N)^c &= (\lambda x : UA^{ty}.M^c)\ \texttt{thunk}\ N^c \\ &\sqsupseteq\sqsubseteq (M^c)[\texttt{thunk}\ N^c/x] \\ &\sqsupseteq\sqsubseteq (M[N/x])^c\end{aligned} \tag{6.13}$$

- $\pi_i(M_1, M_2) \mapsto^n M_i$

$$\begin{aligned}\pi_i(M_1, M_2)^c &= \pi_i\{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\} \\ &\sqsupseteq\sqsubseteq M_i^c\end{aligned} \tag{6.14}$$

- $\texttt{if true then}\ N_1\ \texttt{else}\ N_2 \mapsto^n N_1$

$$\begin{aligned}(\texttt{if true then}\ N_1\ \texttt{else}\ N_2)^c &= z \leftarrow \texttt{ret inl}\ ();\texttt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\} \\ &\sqsupseteq\sqsubseteq \texttt{case inl}\ ()\{x_1.N_1^c \mid x_2.N_2^c\} \\ &\sqsupseteq\sqsubseteq N_1^c\end{aligned} \tag{6.15}$$

- $\texttt{if false then}\ N_1\ \texttt{else}\ N_2 \mapsto^n N_2$, similar to previous.

Next, we have the interesting cases, those specific to gradual type casts/GTT.

- $\langle ? \Leftarrow ? \rangle M \mapsto^n M$:

$$\begin{aligned}(\langle ? \Leftarrow ? \rangle M)^c &= \texttt{force}\ \langle\langle U\underline{?} \swarrow U\underline{?} \rangle\rangle \langle U\underline{?} \nwarrow U\underline{?} \rangle (\texttt{thunk}\ M^c) \\ &\sqsupseteq\sqsubseteq \texttt{force thunk}\ M^c \qquad \text{(Theorem 72, twice)} \\ &\sqsupseteq\sqsubseteq M^c \qquad (U\beta)\end{aligned}$$

- $\langle ? \Leftarrow A \rangle M \mapsto^n \langle ? \Leftarrow G \rangle \langle G \Leftarrow A \rangle M$ where $A \sqsubseteq G$

$$
\begin{aligned}
&\langle ? \Leftarrow A \rangle M^c \\
&= \texttt{force}\ \langle\!\langle UA^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UA^{ty} \swarrow UG^{ty} \rangle\!\rangle \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UA^{ty} \swarrow UG^{ty} \rangle\!\rangle \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow UG^{ty} \rangle \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UA^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow UG^{ty} \rangle \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UA^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow UG^{ty} \rangle (\texttt{thunk}\ (\texttt{force}\ \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c))) \\
&\sqsupseteq\sqsubseteq (\langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle M)^c
\end{aligned}
\tag{6.16}
$$

- $\langle A \Leftarrow ? \rangle V \mapsto^n \langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle V$: similar to previous case.
- $\langle G \Leftarrow ? \rangle \langle G \Leftarrow ? \rangle M \mapsto^n M$

$$
\begin{aligned}
&(\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle M)^c \\
&= \texttt{force}\ \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ (\texttt{force}\ \langle\!\langle U_{\underline{¿}} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c))) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle (\texttt{thunk}\ (\texttt{force}\ \langle U_{\underline{¿}} \nwarrow U_{\underline{¿}} \rangle (\texttt{thunk}\ M^c))) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\!\langle UG^{ty} \swarrow U_{\underline{¿}} \rangle\!\rangle \langle U_{\underline{¿}} \nwarrow UG^{ty} \rangle (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \texttt{force}\ (\texttt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq M^c
\end{aligned}
\tag{6.17}
$$

- $\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle M \mapsto^n \lambda x : A_1'.\langle A_2' \Leftarrow A_2\rangle(M\ (\langle A_1 \Leftarrow A_1'\rangle x))$

$$
\begin{aligned}
&(\langle A_1' \to A_2' \Leftarrow A_1 \to A_2\rangle M)^c \\
&\sqsupseteq\sqsubseteq \mathtt{force}\ \langle\!\langle U(UA_1'^{tv} \to A_2'^{tv})\ \swarrow\ U\underline{\iota}\rangle\!\rangle\langle U(UA_1^{tv} \to A_2^{tv})\ \nwarrow\ U\underline{\iota}\rangle \mathtt{thunk}\ (M^c) \\
&\sqsupseteq\sqsubseteq \mathtt{force}\ \langle\!\langle U(UA_1'^{tv} \to A_2'^{tv})\ \swarrow\ U(U\underline{\iota} \to \underline{\iota})\rangle\!\rangle\langle U(U\underline{\iota} \to \underline{\iota})\ \nwarrow\ U(UA_1^{tv} \to A_2^{tv})\rangle \\
&\qquad (\mathtt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \langle UA_1'^{tv} \to A_2'^{tv}\ \swarrow\ U\underline{\iota} \to \underline{\iota}\rangle \mathtt{force}\ \langle U(U\underline{\iota} \to \underline{\iota})\ \nwarrow\ U(UA_1^{tv} \to A_2^{tv})\rangle(\mathtt{thunk}\ M^c) \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \begin{pmatrix}(\mathtt{force}\ \langle U(U\underline{\iota} \to \underline{\iota})\ \nwarrow\ U(UA_1^{tv} \to A_2^{tv})\rangle(\mathtt{thunk}\ M^c)) \\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)\end{pmatrix} \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \\
&\qquad \left(\begin{array}{l}(\mathtt{force}\ \mathtt{thunk}\ (\lambda y'.\begin{pmatrix}y \leftarrow \langle \underline{F}UA_1^{tv}\ \swarrow\ \underline{F}U\underline{\iota}\rangle(\mathtt{ret}\ y'); \\ \mathtt{force}\ (\langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle \mathtt{thunk}\ ((\mathtt{force}\ \mathtt{thunk}\ M^c)\ y))\end{pmatrix})) \\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)\end{array}\right) \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \\
&\qquad \left(\begin{array}{l}(\lambda y'.\begin{pmatrix}y \leftarrow \langle \underline{F}UA_1^{tv}\ \swarrow\ \underline{F}U\underline{\iota}\rangle(\mathtt{ret}\ y'); \\ \mathtt{force}\ (\langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle(\mathtt{thunk}\ (M^c\ y)))\end{pmatrix}) \\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)\end{array}\right) \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \\
&\qquad \left(\begin{array}{l}(\lambda y'.\begin{pmatrix}y \leftarrow \mathtt{ret}\ \mathtt{thunk}\ (\langle A_1^{tv}\ \swarrow\ \underline{\iota}\rangle(\mathtt{force}\ y')); \\ \mathtt{force}\ (\langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle(\mathtt{thunk}\ (M^c\ y)))\end{pmatrix}) \\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)\end{array}\right) \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \\
&\qquad \left(\begin{array}{l}(\lambda y'.\begin{pmatrix}\mathtt{force}\ (\langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle(\mathtt{thunk}\ (M^c\ (\mathtt{thunk}\ (\langle A_1^{tv}\ \swarrow\ \underline{\iota}\rangle(\mathtt{force}\ y'))))))\end{pmatrix}) \\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)\end{array}\right) \\
&\sqsupseteq\sqsubseteq \lambda x.\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle \mathtt{force}\ \langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle \\
&\qquad \begin{pmatrix}(\mathtt{thunk}\ (M^c\ (\mathtt{thunk}\ (\langle A_1^{tv}\ \swarrow\ \underline{\iota}\rangle(\mathtt{force}\ (\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x))))))\end{pmatrix} \\
&\sqsupseteq\sqsubseteq \lambda x.\mathtt{force}\ \langle\!\langle A_2'^{tv}\ \swarrow\ \underline{\iota}\rangle\!\rangle\langle U\underline{\iota}\ \nwarrow\ UA_2^{tv}\rangle \\
&\qquad \begin{pmatrix}(\mathtt{thunk}\ (M^c\ (\mathtt{thunk}\ (\mathtt{force}\ \langle\!\langle UA_1^{tv}\ \swarrow\ U\underline{\iota}\rangle\!\rangle\langle U\underline{\iota}\ \nwarrow\ UA_1'^{tv}\rangle x)))))\end{pmatrix} \\
&= (\lambda x.\langle A_2' \Leftarrow A_2\rangle(M\ (\langle A_1 \Leftarrow A_1'\rangle x)))^c
\end{aligned}
\tag{6.18}
$$

- $\langle A_1' \times A_2' \Leftarrow A_1 \times A_2 \rangle M \mapsto^n (\langle A_1' \Leftarrow A_1 \rangle \pi_1 M, \langle A_2' \Leftarrow A_2 \rangle \pi_2 M)$

$$
\begin{aligned}
&(\langle A_1' \times A_2' \Leftarrow A_1 \times A_2 \rangle M)^c \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U\underline{¿}\rangle\rangle\langle U(A_1^{ty} \,\&\, A_2^{ty}) \nwarrow U\underline{¿}\rangle \texttt{thunk}\ M^c \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle\rangle\langle U(A_1^{ty} \,\&\, A_2^{ty}) \nwarrow U(\underline{¿} \,\&\, \underline{¿})\rangle \texttt{thunk}\ M^c \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle\rangle \texttt{thunk} \\
&\qquad \{\pi \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 \texttt{force thunk}\ M^c) \\
&\qquad \mid \pi' \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 \texttt{force thunk}\ M^c)\} \\
&\sqsupseteq\sqsubseteq \texttt{force}\ \langle\langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle\rangle \texttt{thunk} \\
&\qquad \{\pi \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\
&\qquad \mid \pi' \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\} \\
&\sqsupseteq\sqsubseteq \langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle \texttt{force thunk} \\
&\qquad \{\pi \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\
&\qquad \mid \pi' \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\} \\
&\sqsupseteq\sqsubseteq \langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle \\
&\qquad \{\pi \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\
&\qquad \mid \pi' \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\} \\
&\sqsupseteq\sqsubseteq \langle U(A_1'^{ty} \,\&\, A_2'^{ty}) \swarrow U(\underline{¿} \,\&\, \underline{¿})\rangle \\
&\qquad \{\pi \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\
&\qquad \mid \pi' \mapsto \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\} \\
&\sqsupseteq\sqsubseteq \begin{array}{l}\{\pi \mapsto \langle A_1'^{ty} \swarrow \underline{¿}\rangle \texttt{force}\ \langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\ \mid \pi' \mapsto \langle A_2'^{ty} \swarrow \underline{¿}\rangle \texttt{force}\ \langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\}\end{array} \\
&\sqsupseteq\sqsubseteq \begin{array}{l}\{\pi \mapsto \texttt{force}\ \langle\langle UA_1'^{ty} \swarrow U\underline{¿}\rangle\rangle\langle U\underline{¿} \nwarrow UA_1^{ty}\rangle \texttt{thunk}\ (\pi_1 M^c) \\ \mid \pi' \mapsto \texttt{force}\ \langle\langle UA_2'^{ty} \swarrow U\underline{¿}\rangle\rangle\langle U\underline{¿} \nwarrow UA_2^{ty}\rangle \texttt{thunk}\ (\pi_2 M^c)\}\end{array} \\
&= (\langle A_1' \Leftarrow A_1 \rangle \pi_1 M, \langle A_2' \Leftarrow A_2 \rangle \pi_2 M)^c
\end{aligned}
\tag{6.19}
$$

□

6.3 LAZY

Next, we consider *lazy* evaluation order, which exhibits much of the behavior of a Haskell-like system, which is like call-by-name in that variables denote delayed computations, but is also like call-by-value in that types have a non-trivial notion of value, so unlike in call-by-name the difference between $\mho$ and $\lambda x.\mho$ is observable. In Haskell this is observable using the *seq* operator `seq` M N which when forced reduces

M to a value and then forces N. In our calculus we capture this by making the let-binding construct strict $\texttt{let}\ x = M; N$ will evaluate M to a value and then bind a trivial thunk that returns that value to x before evaluating N. In Haskell, this strict let is not necessary because of Haskell's stateful call-by-need semantics, but note here that since the only effects in our language are divergence and an uncatchable error (like Haskell), the difference between call-by-need and lazy evaluation is not extensional and is instead relevant only for evaluation of space-usage. While this is of course an important consideration, it is not necessary to capture this to study the extensional behavior of casts, and we show that by elaboration to GTT, previous designs for contracts in Haskell can be reproduced.

In particular, in Haskell/lazy evaluation, the call-by-name reduction for the function type is incorrect:

$$E[(\langle A \to B \Leftarrow A' \to B' \rangle M)] \mapsto E[\lambda x.\langle B \Leftarrow B' \rangle (M\ (\langle A' \Leftarrow A \rangle x))] \quad \text{(wrong)}$$

The reason is that the scrutinee on the right hand side is a λ, whereas the left hand side might not be. This results in a violation of what Dagand and Thiemann call "meaning preservation": adding a contract changes the termination behavior of a term. In the terminology of gradual typing, we would say that this is a violation of *graduality*, adding a cast resulted in divergence being replaced by termination, but graduality only allows the behavior to stay the same or introduce an *error*. The correct definition is that the cast should be *strict*, essentially the same as the CBV definition:

$$E[(\langle A \to B \Leftarrow A' \to B' \rangle V)] \mapsto E[\lambda x.\langle B \Leftarrow B' \rangle (V\ (\langle A' \Leftarrow A \rangle x))]$$

And we will show that this behavior arises naturally from Levy's lazy translation into CBPV.

Values, evaluation contexts and operational semantics for lazy evaluation are defined in Figure 6.8. Values are λs for functions, pairs of arbitrary terms for the lazy product, true and false for booleans, and tagged but unevaluated terms for the dynamic type. Evaluation contexts are the same as CBN except adding casts.

Next, we define the operational semantics. First, let-binding, as mentioned above is strict, while function application is non-strict in its argument. This difference is what distinguishes it from CBN and invalidates the strong η principle for functions and products. Projection is lazy like call-by-name. Then we have the cast reductions, which are the same as CBV, making it even more clear that this evaluation order is something of a mix of CBN and CBV.

Next, we have the translation into GTT, which follows Levy's definition. This is the most complicated of the three elaborations. First, types denote value types since they have a notion of value. However, variables $x : A$ are not elaborated directly to variables of the elaborated type, but instead thunks of computations that return values

$$
\begin{array}{lll}
V & ::= & \lambda x.M \mid (M_1, M_2) \mid \texttt{true} \mid \texttt{false} \mid \langle ? \Leftarrow G \rangle M \\
E & ::= & [\cdot] \mid \texttt{let } x = E; N \mid E\ M \mid \pi_i E \mid \texttt{if } E \texttt{ then } M_t \texttt{ else } M_f \mid \langle A \Leftarrow B \rangle E
\end{array}
$$

$$
\begin{array}{ll}
E[\texttt{let } x = V; N] & \mapsto^l E[N[V/x]] \\
E[(\lambda x.M)N] & \mapsto^l E[M[N/x]] \\
E[\pi_i\ (M_1, M_2)] & \mapsto^l E[M_i] \\
E[\texttt{if true then } M_t \texttt{ else } M_f] & \mapsto^l E[M_t] \\
E[\texttt{if false then } M_t \texttt{ else } M_f] & \mapsto^l E[M_f]
\end{array}
$$

?ID
$$E[\langle ? \Leftarrow ? \rangle V] \mapsto^l E[V]$$

DECOMPUP
$$\frac{A \sqsubseteq G}{E[\langle ? \Leftarrow A \rangle M] \mapsto^l E[\langle ? \Leftarrow G \rangle \langle G \Leftarrow A \rangle M]}$$

DECOMPDN
$$\frac{A \sqsubseteq G}{E[\langle A \Leftarrow ? \rangle M] \mapsto^l E[\langle A \Leftarrow G \rangle \langle G \Leftarrow ? \rangle M]}$$

TAGMATCH
$$E[\langle G \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle M] \mapsto^l E[M]$$

TAGMISMATCH
$$\frac{G \neq G'}{E[\langle G' \Leftarrow ? \rangle \langle ? \Leftarrow G \rangle M] \mapsto^l \mathcal{U}}$$

EILLY
$$\frac{A \sqsubseteq G_A \qquad B \sqsubseteq G_B \qquad G_A \neq G_B}{E[\langle B \Leftarrow A \rangle M] \mapsto^l \mathcal{U}}$$

$$
\begin{array}{ll}
E[(\langle A \to B \Leftarrow A' \to B' \rangle V)] & \mapsto^l E[\lambda x.\langle B \Leftarrow B' \rangle (V\ (\langle A' \Leftarrow A \rangle x))] \\
E[(\langle A_1 \times A_2 \Leftarrow A_1' \times A_2' \rangle V)] & \mapsto^l E[(\langle A_1 \Leftarrow A_1' \rangle (\pi_1 V), \langle A_2 \Leftarrow A_2' \rangle (\pi_2 V))]
\end{array}
$$

Figure 6.8: Lazy Reduction

If $\Gamma \vdash M : A$, then $UF\Gamma^{ty} \vdash M^c : FA^{ty}$

$$
\begin{aligned}
?^{ty} &= ? \\
(A \to A')^{ty} &= U(UFA^{ty} \to FA'^{ty} \\
(A_1 \times A_2)^{ty} &= U(FA_1^{ty} \,\&\, FA_2^{ty}) \\
\texttt{Bool}^{ty} &= 1 + 1
\end{aligned}
$$

$$
\begin{aligned}
x^c &= \texttt{force}\ x \\
(\texttt{let}\ x = M; N)^c &= y \leftarrow M^c; \texttt{let}\ x = \texttt{thunk}\ \texttt{ret}\ y; N^c \\
(\langle A_2 \Leftarrow A_1 \rangle M)^c &= \langle FA_2^{ty} \swarrow F? \rangle \langle\!\langle F? \nwarrow FA_1^{ty} \rangle\!\rangle [M^c] \\
(\lambda x : A.M)^c &= \texttt{ret}\ \texttt{thunk}\ (\lambda x : UFA^{ty}.M^c) \\
M\ N^c &= f \leftarrow M^c;\ \texttt{force}\ f\ (\texttt{thunk}\ N^c) \\
(M_1, M_2)^c &= \texttt{ret}\ \texttt{thunk}\ (\{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\}) \\
\pi_i M^c &= p \leftarrow M^c; \pi_i \texttt{force}\ p \\
\texttt{true}^c &= \texttt{ret}\ \texttt{inl}\ () \\
\texttt{false}^c &= \texttt{ret}\ \texttt{inr}\ () \\
(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2)^c &= z \leftarrow M^c; \texttt{case}\ z\{x_1.\ N_1^c \mid x_2.\ N_2^c\}
\end{aligned}
\tag{6.20}
$$

Figure 6.9: Lazy to GTT translation

of that type UFA^{ty}. The function type translation illustrates this. A value of product type is a thunk that when forced can be projected as either A_i-returning computation. Note that an equivalent definition would be to say that a product value is a pair of thunks that return A_is: $(A_1 \times A_2)^{ty} \cong UFA_1^{ty} \times UFA_i^{ty}$, the two definitions are isomorphic in CBPV. Then the term translation is something like a combination of CBV and CBN translations: following the structure of the CBV translation, but inserting thunks in the appropriate place like CBN.

To reason about plugging in evaluation contexts in the correctness proofs, we additionally define a *stack* translation that directly translates lazy evaluation contexts to GTT stacks in Figure 6.4

We then prove a few correctness principles for these with respect to the term translation.

Lemma 102. $(M[N/x])^c \sqsupseteq\sqsubseteq M^c[\texttt{thunk}\ (N^c)/x]$

Proof. The proof is the same as the CBN proof. □

Lemma 103. $V^c \sqsupseteq\sqsubseteq \texttt{ret}\ V^v$

Proof. By case analysis on V, trivial in all cases except the cast case, which follows the same argument as the CBV case. □

Lemma 104. $(E[M])^c \sqsupseteq\sqsubseteq E^s[M^c]$

If $\cdot \vdash V : A$ then $\cdot \vdash V^v : A^{ty}$
If $\bullet : A \vdash E : A'$ then $\bullet : FA^{ty} \vdash E^s : \underline{F}A'^{ty}$

$$
\begin{aligned}
\langle ? \Leftarrow G \rangle V^v &= \langle ? \nwarrow G \rangle V^v \\
(\lambda x.M)^v &= \mathtt{thunk}\ (\lambda x.M^c) \\
(M_1, M_2)^v &= \mathtt{thunk}\ (\{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\}) \\
\mathtt{true}^v &= \mathtt{inl}\ () \\
\mathtt{false}^v &= \mathtt{inr}\ ()
\end{aligned}
$$

$$
\begin{aligned}
\bullet^s &= \bullet \\
(\mathtt{let}\ x = E; N)^s &= y \leftarrow E^s; \mathtt{let}\ x = \mathtt{ret}\ \mathtt{thunk}\ y; N^c \\
(\langle A \Leftarrow B \rangle E)^s &= x \leftarrow E^s; \langle FA_2^{ty} \swarrow F? \rangle (\mathtt{ret}\ \langle ? \nwarrow A_1^{ty} \rangle x) \\
(E\ N)^s &= f \leftarrow E^s; (\mathtt{force}\ f)\ (\mathtt{thunk}\ N^c) \\
(\pi_i E)^s &= p \leftarrow E^s; \pi_i \mathtt{force}\ p \\
(\mathtt{if}\ E\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2)^s &= z \leftarrow E^s; \mathtt{case}\ z\{x_1.N_1^c \mid x_2.N_2^c\}
\end{aligned}
\tag{6.21}
$$

Figure 6.10: Lazy Value and Stack translation

Proof. By induction on E. All cases are straightforward by congruence and induction hypothesis. □

And the central theorem is easily established.

Theorem 105. *If $M \mapsto^l N$ by any rule except* TagMismatch *or* Silly, *then $M^c \sqsupseteq\sqsubseteq N^c$.*

Proof. In all cases, by Lemma 97, congruence and $E[\mho] \sqsupseteq\sqsubseteq \mho$, it is sufficient to consider the case that $E = \bullet$.

First, we have the cases not involving casts.

- $\mathtt{let}\ x = V; N \mapsto^l N[V/x]$

$$
\begin{aligned}
(\mathtt{let}\ x = V; N)^c &= y \leftarrow V^c; \mathtt{let}\ x = \mathtt{thunk}\ \mathtt{ret}\ y; N^c \\
&\sqsupseteq\sqsubseteq y \leftarrow \mathtt{ret}\ ; \mathtt{let}\ x = \mathtt{thunk}\ \mathtt{ret}\ y; N^c \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ x = \mathtt{thunk}\ \mathtt{ret}\ V^v; N^c \\
&\sqsupseteq\sqsubseteq N^c[\mathtt{thunk}\ \mathtt{ret}\ V^v/x] \\
&\sqsupseteq\sqsubseteq N^c[\mathtt{thunk}\ V^c/x] \\
&\sqsupseteq\sqsubseteq (N[V/x]^c)
\end{aligned}
\tag{6.22}
$$

- $(\lambda x : A.M)\, N \mapsto^l M[N/x]$

$$\begin{aligned}((\lambda x : A.M)\ N)^c &= f \leftarrow \texttt{ret thunk}\ (\lambda x : UFA^{ty}.M^c); (\texttt{force}\ f)\ \texttt{thunk}\ N^c \\ &\sqsupseteq\sqsubseteq (\texttt{force}\ (\texttt{thunk}\ (\lambda x : UFA^{ty}.M^c)))\ \texttt{thunk}\ N^c \\ &\sqsupseteq\sqsubseteq (\lambda x : UFA^{ty}.M^c)\ \texttt{thunk}\ N^c \\ &\sqsupseteq\sqsubseteq M^c[\texttt{thunk}\ N^c/x] \\ &\sqsupseteq\sqsubseteq (M[N/x])^c\end{aligned} \tag{6.23}$$

- $\pi_i(M_1, M_2) \mapsto^l M_i$

$$\begin{aligned}\pi_i(M_1, M_2)^c &= p \leftarrow \texttt{ret thunk}\ \{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\}; \pi_i \texttt{force}\ p \\ &\sqsupseteq\sqsubseteq \pi_i \texttt{force}\ (\texttt{thunk}\ \{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\}) \\ &\sqsupseteq\sqsubseteq \pi_i\{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\} \\ &\sqsupseteq\sqsubseteq \pi_i\{\pi \mapsto M_1^c \mid \pi' \mapsto M_2^c\} \\ &\sqsupseteq\sqsubseteq M_i^c\end{aligned} \tag{6.24}$$

- $\texttt{if true then}\ N_1\ \texttt{else}\ N_2 \mapsto^l N_1$, the same argument as CBV and CBN.
- $\texttt{if false then}\ N_1\ \texttt{else}\ N_2 \mapsto^l N_2$, similar to previous.

Next, the cast cases follow by the same argument as the CBV case. □

7

MODELS

To show the soundness of GTT as a theory, and demonstrate its relationship to operational definitions of observational equivalence and the gradual guarantee we used in Chapters 2, 3, we develop *models* of GTT using observational error approximation of a *non-gradual* CBPV calculus. Analogous to our construction in Chapter 3, we will elaborate GTT into a typed language, in our case CBPV with recursive types and errors. Then we will show that giving a semantics of the term ordering $\sqsubseteq$ as error approximation results in a *model* of the theory of GTT in that every syntactic proof in GTT of ordering and equivalence holds true of the elaboration into CBPV.

We call this the *contract translation* because it translates the built-in casts of the gradual language into ordinary terms implemented in a non-gradual language. While contracts are typically implemented in a dynamically typed language, our target is typed, retaining type information similarly to manifest contracts [32]. We give implementations of the dynamic value type in the usual way as a recursive sum of basic value types, i.e., using type tags, and we give implementations of the dynamic computation type as the dual: a recursive product of basic computation types.

Writing $[\![M]\!]$ for any of the contract translations, the remaining sections of the paper establish:

Theorem 106 (Equi-precision implies Observational Equivalence)**.** *If* $\Gamma \vdash M_1 \sqsupseteq\sqsubseteq M_2 : \underline{B}$*, then for any closing GTT context* $C : (\Gamma \vdash \underline{B}) \Rightarrow (\cdot \vdash \underline{F}(1+1))$*,* $[\![C[M_1]]\!]$ *and* $[\![C[M_2]]\!]$ *have the same behavior: both diverge, both run to an error, or both run to* `true` *or both run to* `false`*.*

Theorem 107 (Graduality)**.** *If* $\Gamma_1 \sqsubseteq \Gamma_2 \vdash M_1 \sqsubseteq M_2 : B_1 \sqsubseteq B_2$*, then for any GTT context* $C : (\Gamma_1 \vdash B_1) \Rightarrow (\cdot \vdash \underline{F}(1+1))$*, and any valid interpretation of the dynamic types, either*

1. $[\![C[M_1]]\!] \Downarrow \mho$*, or*
2. $[\![C[M_1]]\!] \Uparrow$ *and* $[\![C[\langle B_1 \swarrow B_2 \rangle M_2[\langle \Gamma_2 \nwarrow \Gamma_1 \rangle \Gamma_1]]]\!] \Uparrow$*, or*
3. $[\![C[M_1]]\!] \Downarrow \texttt{ret}\ V$*,* $[\![C[\langle B_1 \swarrow B_2 \rangle M_2[\langle \Gamma_2 \nwarrow \Gamma_1 \rangle \Gamma_1]]]\!] \Downarrow \texttt{ret}\ V$*, and* $V = \texttt{true}$ *or* $V = \texttt{false}$*.*

As a consequence we will also get consistency of our logic of precision:

Corollary 108 (Consistency of GTT)**.** $\cdot \vdash \texttt{ret true} \sqsubseteq \texttt{ret false} : \underline{F}(1+1)$ *is not provable in GTT.*

Proof. They are distinguished by the identity context. □

We break down this proof into 3 major steps.

1. (This section) We translate GTT into a statically typed CBPV* language where the casts of GTT are translated to "contracts" in GTT: i.e., CBPV terms that implement the runtime type checking. We translate the term precision of GTT to an inequational theory for CBPV. Our translation is parameterized by the implementation of the dynamic types, and we demonstrate two valid implementations, one more direct and one more Scheme-like.

2. (§7.3) Next, we eliminate all uses of complex values and stacks from the CBPV language. We translate the complex values and stacks to terms with a proof that they are "pure" (thunkable or linear [53]). This part has little to do with GTT specifically, except that it shows the behavioral property that corresponds to upcasts being complex values and downcasts being complex stacks.

3. (§7.4.3) Finally, with complex values and stacks eliminated, we give a standard operational semantics for CBPV and define a *logical relation* that is sound and complete with respect to observational error approximation. Using the logical relation, we show that the inequational theory of CBPV is sound for observational error approximation.

By composing these, we get a model of GTT where equiprecision is sound for observational equivalence and an operational semantics that satisfies the graduality theorem.

7.1 CALL-BY-PUSH-VALUE

Next, we define the call-by-push-value language CBPV* that will be the target for our contract translations of GTT. CBPV* is the axiomatic version of call-by-push-value *with* complex values and stacks, while CBPV (§7.3) will designate the operational version of call-by-push-value with only operational values and stacks. CBPV* is almost a subset of GTT obtained as follows: We remove the casts and the dynamic types ? and ¿ (the shaded pieces) from the syntax and typing rules in Figures 5.1 and 5.2. There is no type precision, and the inequational theory of CBPV* is the homogeneous fragment of term precision in Figure 5.4 and Figure 5.5 (judgements $\Gamma \vdash E \sqsubseteq E' : T$ where $\Gamma \vdash E, E' : T$, with all the same rules in that figure thus restricted). The inequational axioms are the Type Universal Properties ($\beta\eta$ rules) and Error Properties (with ErrBot made homogeneous) from Figure 5.6. To implement the casts and dynamic types, we *add* general *recursive* value types ($\mu X.A$, the fixed point of X val type $\vdash A$ val type) and *corecursive* computation types ($\nu \underline{Y}.\underline{B}$, the fixed point of $\underline{Y}$ comp type $\vdash \underline{B}$ comp type). The

recursive type $\mu X.A$ is a value type with constructor `roll`, whose eliminator is pattern matching, whereas the corecursive type $\nu \underline{Y}.\underline{B}$ is a computation type defined by its eliminator (`unroll`), with an introduction form that we also write as `roll`. We extend the inequational theory with monotonicity of each term constructor of the recursive types, and with their $\beta\eta$ rules. In Figure 7.1, we write **+** ::= and **-** ::= to indicate the diff from the grammar in Figure 5.1.

Value Types	A	**+** ::=	$\mu X.A \mid X$
		- ::=	$?$
Computation Types	$\underline{B}$	**+** ::=	$\nu \underline{Y}.\underline{B} \mid \underline{Y}$
		- ::=	$¿\underline{}$
Values	V	**+** ::=	$\mathtt{roll}_{\mu X.A}\ V$
		- ::=	$\langle A \nwarrow A \rangle V$
Terms	M	**+** ::=	$\mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M \mid \mathtt{unroll}\ M$
	M	**-** ::=	$\langle \underline{B} \swarrow \underline{B} \rangle M$
Both	E	**+** ::=	$\mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.E$

$$\frac{\Gamma \vdash V : A[\mu X.A/X]}{\Gamma \vdash \mathtt{roll}_{\mu X.A}\ V : \mu X.A}\ \mu\mathrm{I} \qquad \frac{\Gamma \vdash V : \mu X.A \quad \Gamma, x : A[\mu X.A/X] \mid \Delta \vdash E : T}{\Gamma \mid \Delta \vdash \mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.E : T}\ \mu\mathrm{E}$$

$$\frac{\Gamma \mid \Delta \vdash M : \underline{B}[\nu \underline{Y}.\underline{B}]}{\Gamma \mid \Delta \vdash \mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M : \nu \underline{Y}.\underline{B}}\ \nu\mathrm{I}$$

$$\frac{\Gamma \mid \Delta \vdash M : \nu \underline{Y}.\underline{B}}{\Gamma \mid \Delta \vdash \mathtt{unroll}\ M : \underline{B}[\nu \underline{Y}.\underline{B}]}\ \nu\mathrm{E} \qquad \frac{\Gamma \vdash V \sqsubseteq V' : A[\mu X.A/X]}{\Gamma \vdash \mathtt{roll}\ V \sqsubseteq \mathtt{roll}\ V' : \mu X.A}\ \mu\mathrm{ICong}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \mid \Delta \vdash E \sqsubseteq E' : T}{\Gamma \mid \Delta \vdash \mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.E \sqsubseteq \mathtt{unroll}\ V'\ \mathtt{to}\ \mathtt{roll}\ x.E' : T}\ \mu\mathrm{ECong}$$

$$\frac{\Gamma \mid \Delta \vdash M \sqsubseteq M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \mid \Delta \vdash \mathtt{roll}\ M \sqsubseteq \mathtt{roll}\ M' : \nu \underline{Y}.\underline{B}}\ \nu\mathrm{ICong}$$

$$\frac{\Gamma \mid \Delta \vdash M \sqsubseteq M' : \nu \underline{Y}.\underline{B}}{\Gamma \mid \Delta \vdash \mathtt{unroll}\ M \sqsubseteq \mathtt{unroll}\ M' : \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}\ \nu\mathrm{ECong}$$

Recursive Type Axioms

Figure 7.1: CBPV* types, terms, recursive types (diff from GTT)

Type	β	η
μ	$\texttt{unroll roll}\ V\ \texttt{to roll}\ x.E \sqsupseteq\sqsubseteq E[V/x]$	$E \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ y.E[\texttt{roll}\ y/x]$ where $x : \mu X.A \vdash E : T$
ν	$\texttt{unroll roll}\ M \sqsupseteq\sqsubseteq M$	$\bullet : \nu \underline{Y}.\underline{B} \vdash \bullet \sqsupseteq\sqsubseteq \texttt{roll unroll} \bullet : \nu \underline{Y}.\underline{B}$

Figure 7.2: CBPV* $\beta\eta$ rules (recursive types)

7.2 ELABORATING GTT

As shown in Theorems 72, 75, 76, almost all of the contract translation is uniquely determined already. However, the interpretation of the dynamic types and the casts between the dynamic types and ground types G and $\underline{G}$ are not determined (they were still postulated in Lemma 81). For this reason, our translation is *parameterized* by an interpretation of the dynamic types and the ground casts. By Theorems 73, 74, we know that these must be *embedding-projection pairs* (ep pairs), which we now define in CBPV*. There are two kinds of ep pairs we consider: those between value types (where the embedding models an upcast) and those between computation types (where the projection models a downcast).

Definition 109 (Value and Computation Embedding-Projection Pairs)**.**

1. A *value ep pair* from A to A' consists of an *embedding* value $x : A \vdash V_e : A'$ and *projection* stack $\bullet : \underline{F}A' \vdash S_p : \underline{F}A$, satisfying the *retraction* and *projection* properties:

$$x : A \vdash \texttt{ret}\ x \sqsupseteq\sqsubseteq S_p[\texttt{ret}\ V_e] : \underline{F}A \qquad \bullet : \underline{F}A' \vdash x \leftarrow S_p; \texttt{ret}\ V_e \sqsubseteq \bullet : \underline{F}A'$$

2. A *computation ep pair* from $\underline{B}$ to $\underline{B}'$ consists of an *embedding* value $z : U\underline{B} \vdash V_e : U\underline{B}'$ and a *projection* stack $\bullet : \underline{B}' \vdash S_p : \underline{B}$ satisfying *retraction* and *projection* properties:

$$z : U\underline{B} \vdash \texttt{force}\ z \sqsupseteq\sqsubseteq S_p[\texttt{force}\ V_e] : \underline{B} \quad w : U\underline{B}' \vdash V_e[\texttt{thunk}\ S_p[\texttt{force}\ w]] \sqsubseteq w : U\underline{B}'$$

While this formulation is very convenient in that both kinds of ep pairs are pairs of a value and a stack, the projection properties are often occur more naturally in the following forms:

Lemma 110 (Alternative Projection)**.** *If (V_e, S_p) is a value ep pair from A to A' and $\Gamma, y : A' \mid \Delta \vdash M : \underline{B}$, then*

$$\Gamma, x' : A' \vdash x \leftarrow S_p[\texttt{ret}\ x']; M[V_e/y] \sqsubseteq M[x'/y]$$

Similarly, if (V_e, S_p) is a computation ep pair from $\underline{B}$ to $\underline{B}'$, and $\Gamma \vdash M : \underline{B}'$ then

$$\Gamma \vdash V_e[\texttt{thunk}\ S_p[M]] \sqsubseteq \texttt{thunk}\ M : U\underline{B}'$$

Proof. For the first,

$$\begin{aligned} x \leftarrow S_p[\mathtt{ret}\ x']; M[V_e/y] &\sqsupseteq\sqsubseteq y \leftarrow (x \leftarrow S_p[\mathtt{ret}\ x']; \mathtt{ret}\ V_e); M && \text{(comm conv, } \underline{F}\beta\text{)} \\ y \leftarrow \mathtt{ret}\ x'; M && \text{(projection)} \\ M[x'/y] && (\underline{F}\beta) \end{aligned}$$

For the second,

$$\begin{aligned} V_e[\mathtt{thunk}\ S_p[M]] &\sqsupseteq\sqsubseteq V_e[\mathtt{thunk}\ S_p[\mathtt{force}\ \mathtt{thunk}\ M]] && (U\beta) \\ &\sqsubseteq \mathtt{thunk}\ M && \text{(projection)} \end{aligned}$$

□

Using our definition of ep pairs, and using the notion of ground type from §5.3.5 *with* 0 *and* ⊤ *removed*, we define

Definition 111 (Dynamic Type Interpretation)**.** A ?, ¿ interpretation ρ consists of (1) a CBPV value type $\rho(?)$, (2) a CBPV computation type $\rho(\underline{¿})$, (3) for each value ground type G, a value ep pair $(x.\rho_e(G), \rho_p(G))$ from $[\![G]\!]_\rho$ to $\rho(?)$, and (4) for each computation ground type $\underline{G}$, a computation ep pair $(z.\rho_e(\underline{G}), \rho_p(\underline{G}))$ from $[\![\underline{G}]\!]_\rho$ to $\rho(\underline{¿})$. We write $[\![G]\!]_\rho$ and $[\![\underline{G}]\!]_\rho$ for the interpretation of a ground type, replacing ? with $\rho(?)$, ¿ with $\rho(\underline{¿})$, and compositionally otherwise.

Next, we show several possible interpretations of the dynamic type that will all give, by construction, implementations that satisfy the gradual guarantee. Our interpretations of the value dynamic type are not surprising. They are the usual construction of the dynamic type using type tags: i.e., a recursive sum of basic value types. On the other hand, our interpretations of the computation dynamic type are less familiar. In duality with the interpretation of ?, we interpret ¿ as a recursive *product* of basic computation types. This interpretation has some analogues in previous work on the duality of computation [30, 92], but the most direct interpretation (definition 115) does not correspond to any known work on dynamic/gradual typing. Then we show that a particular choice of which computation types is basic and which are derived produces an interpretation of the dynamic computation type as a type of variable-arity functions whose arguments are passed on the stack, producing a model similar to Scheme without accounting for control effects (definition 120).

7.2.1 *Natural Dynamic Type Interpretation*

Our first dynamic type interpretation is to make the value and computation dynamic types sums and products of the ground value and computation types, respectively. This forms a model of GTT for the following reasons. For the value dynamic type ?, we need a value

embedding (the upcast) from each ground value type G with a corresponding projection. The easiest way to do this would be if for each G, we could rewrite ? as a sum of the values that fit G and some "complement" of G, $?_{\neg G}$ of those that don't–? $\cong G + ?_{\neg G}$. Then we could use the the following fact.

Lemma 112 (Sum Injections are Value Embeddings)**.** *For any A, A', there are value ep pairs from A and A' to $A + A'$ where the embeddings are* `inl` *and* `inr` *.*

Proof. Define the embedding of A to just be $x.\texttt{inl}\ x$ and the projection to be

$$y \leftarrow \bullet; \texttt{case}\ y\{\texttt{inl}\ x.\texttt{ret}\ x \mid \texttt{inr}\ \_.\mathcal{U}\}.$$

We show this satisfies retraction and projection in the supplementary material. □

Proof. This satisfies retraction (using $\underline{F}(+)$ induction (Lemma 113) defined immediately after this proof, `inr` case is the same):

$$\begin{aligned} y \leftarrow \texttt{inl}\ x; \texttt{case}\ y\{\texttt{inl}\ x.\texttt{ret}\ x \mid \texttt{inr}\ \_.\mathcal{U}\} &\sqsupseteq\sqsubseteq \texttt{case}\ \texttt{inl}\ x\{\texttt{inl}\ x.\texttt{ret}\ x \mid \texttt{inr}\ \_.\mathcal{U}\} \\ &\qquad (\underline{F}\beta) \\ &\sqsupseteq\sqsubseteq \texttt{ret}\ x \qquad (+\beta) \end{aligned}$$

and projection (similarly using $\underline{F}(+)$ induction):

$$\begin{aligned} x' : A + A' \vdash (y \leftarrow \texttt{ret}\ x'; \texttt{case}\ y\{\texttt{inl}\ x.\texttt{ret}\ x \mid \texttt{inr}\ \_.\mathcal{U}\}) &\leftarrow x; \texttt{ret}\ \texttt{inl}\ x \\ &\sqsupseteq\sqsubseteq (\texttt{case}\ x'\{\texttt{inl}\ x.\texttt{ret}\ x \mid \texttt{inr}\ \_.\mathcal{U}\}) \leftarrow x; \texttt{ret}\ \texttt{inl}\ x \\ &\qquad (\underline{F}\beta) \\ &\sqsupseteq\sqsubseteq (\texttt{case}\ x'\{\texttt{inl}\ x.x \leftarrow \texttt{ret}\ x; \texttt{ret}\ \texttt{inl}\ x \mid \texttt{inr}\ \_.x \leftarrow \mathcal{U}; \texttt{ret}\ \texttt{inl}\ x\}) \\ &\qquad \text{(commuting conversion)} \\ &\sqsupseteq\sqsubseteq (\texttt{case}\ x'\{\texttt{inl}\ x.\texttt{ret}\ \texttt{inl}\ x \mid \texttt{inr}\ \_.\mathcal{U}\}) \\ &\qquad (\underline{F}\beta, \mathcal{U}\ \text{strictness}) \\ &\sqsubseteq (\texttt{case}\ x'\{\texttt{inl}\ x.\texttt{ret}\ \texttt{inl}\ x \mid \texttt{inr}\ y.\texttt{ret}\ \texttt{inl}\ y\}) \\ &\qquad (\mathcal{U}\ \text{bottom}) \\ &\sqsupseteq\sqsubseteq \texttt{ret}\ x' \qquad (+\eta) \end{aligned}$$

□

Lemma 113 ($\underline{F}(+)$ Induction Principle)**.** $\Gamma \mid \cdot : \underline{F}(A_1 + A_2) \vdash M_1 \sqsubseteq M_2 : \underline{B}$ *holds if and only if* $\Gamma, V_1 : A_1 \vdash M_1[\texttt{ret inl}\ V_1] \sqsubseteq M_2[\texttt{ret inl}\ V_2] : \underline{B}$ *and* $\Gamma, V_2 : A_2 \vdash M_2[\texttt{ret inr}\ V_2] \sqsubseteq M_2[\texttt{ret inr}\ V_2] : \underline{B}$

This shows why the type tag interpretation works: it makes the dynamic type in some sense the minimal type with injections from each G: the sum of all value ground types $? \cong \Sigma_G G$.

The dynamic computation type $\underline{¿}$ can be naturally defined by a dual construction, by the following dual argument. First, we want a computation ep pair from $\underline{G}$ to $\underline{¿}$ for each ground computation

type $\underline{G}$. Specifically, this means we want a stack from $\underline{¿}$ to $\underline{G}$ (the downcast) with an embedding. The easiest way to get this is if, for each ground computation type $\underline{G}$, $\underline{¿}$ is equivalent to a lazy product of $\underline{G}$ and "the other behaviors", i.e., $\underline{¿} \cong \underline{G} \,\&\, \underline{¿}_{-\underline{G}}$. Then the embedding on π performs the embedded computation, but on π' raises a type error. The following lemma, dual to Lemma 112 shows this forms a computation ep pair:

Lemma 114 (Lazy Product Projections are Computation Projections)**.** *For any $\underline{B}, \underline{B}'$, there are computation ep pairs from $\underline{B}$ and $\underline{B}'$ to $\underline{B} \,\&\, \underline{B}'$ where the projections are π and π'.*

Proof. This satisfies retraction:

$$\begin{aligned} \pi(\texttt{force}\ (\texttt{thunk}\ (\texttt{force}\ z, \mho))) &\sqsupseteq\sqsubseteq \pi(\texttt{force}\ z, \mho) && (U\beta) \\ &\sqsupseteq\sqsubseteq \texttt{force}\ z && (\&\beta) \end{aligned}$$

and projection:

$$\begin{aligned} &\texttt{thunk}\ (\texttt{force}\ \texttt{thunk}\ \pi\texttt{force}\ w, \mho) \\ &\sqsupseteq\sqsubseteq \texttt{thunk}\ (\pi\texttt{force}\ w, \mho) && (U\beta) \\ &\sqsubseteq \texttt{thunk}\ (\pi\texttt{force}\ w, \pi'\texttt{force}\ w) && (\mho \text{ bottom}) \\ &\sqsupseteq\sqsubseteq \texttt{thunk force}\ w && (\&\eta) \\ &\sqsupseteq\sqsubseteq w && (U\eta) \end{aligned}$$

□

From this, we see that the easiest way to construct an interpretation of the dynamic computation type is to make it a lazy product of all the ground types $\underline{G}$: $\underline{¿} \cong \&_{\underline{G}}\, \underline{G}$. Using recursive types, we can easily make this a definition of the interpretations:

Definition 115 (Natural Dynamic Type Interpretation)**.** The following defines a dynamic type interpretation. We define the types to satisfy the isomorphisms

$$? \cong 1 + (? \times ?) + (? + ?) + U\underline{¿} \qquad \underline{¿} \cong (\underline{¿} \,\&\, \underline{¿}) \,\&\, (? \to \underline{¿}) \,\&\, \underline{F}?$$

with the ep pairs defined as in Lemma 112 and Lemma 114.

Proof. We can construct $?, \underline{¿}$ explicitly using recursive and corecursive types. Specifically, we make the recursion explicit by defining open versions of the types:

$$\begin{aligned} X, \underline{Y} &\vdash ?_o = 1 + (X \times X) + (X + X) + U\underline{Y} \text{ val type} \\ X, \underline{Y} &\vdash \underline{¿}_o = (\underline{Y} \,\&\, \underline{Y}) \,\&\, (X \to \underline{Y}) \,\&\, \underline{F}X \text{ comp type} \end{aligned}$$

Then we define the types $?, \underline{¿}$ using a standard encoding:

$$\begin{aligned} ? &= \mu X.?_o[\nu \underline{Y}.\underline{¿}_o / \underline{Y}] \\ \underline{¿} &= \nu \underline{Y}.\underline{¿}_o[\mu X.?_o / X] \end{aligned}$$

$$\frac{\Gamma \mid \Delta \vdash V : ? \qquad \Gamma, x_1 : 1 \mid \Delta \vdash E_1 : T \qquad \Gamma, x_\times : ? \times ? \mid \Delta \vdash E_\times : T \qquad \Gamma, x_+ : ? + ? \mid \Delta \vdash E_+ : T \qquad \Gamma, x_U : U\underline{¿} \mid \Delta \vdash E_U : T}{\Gamma \mid \Delta \vdash \mathtt{tycase}\ V\,\{x_1.E_1 \mid x_\times.E_\times \mid x_+.E_+ \mid x_U.E_U\} : T}\ ?\mathrm{E}$$

$$\mathtt{tycase}\ (\langle ? \nwarrow G \rangle V)\,\{x_1.E_1 \mid x_\times.E_\times \mid x_+.E_+ \mid x_U.E_U\} \sqsupseteq\sqsubseteq E_G[V/x_G] \qquad (?\beta)$$

$$\frac{\Gamma, x : ? \mid \Delta \vdash E : \underline{B}}{E \sqsupseteq\sqsubseteq \mathtt{tycase}\ x\,\{x_1.E[\langle ? \nwarrow 1\rangle/x_1] \mid x_\times.E[\langle ? \nwarrow \times\rangle/x_\times] \mid x_+.E[\langle ? \nwarrow +\rangle/x_+] \mid x_U.E[\langle ? \nwarrow U\rangle/x_U]\}}\ ?\eta$$

$$\frac{\Gamma \mid \Delta \vdash M_\to : ? \to \underline{¿} \qquad \Gamma \mid \Delta \vdash M_\& : \underline{¿}\,\&\,\underline{¿} \qquad \Gamma \mid \Delta \vdash M_{\underline{F}} : \underline{F}}{\Gamma \mid \Delta \vdash \{\& \mapsto M_\& \mid (\to) \mapsto M_\to \mid \underline{F} \mapsto M_{\underline{F}}\} : \underline{¿}}\ \underline{¿}$$

$$\langle \underline{G} \swarrow \underline{¿} \rangle \{\& \mapsto M_\& \mid (\to) \mapsto M_\to \mid \underline{F} \mapsto M_{\underline{F}}\} \sqsupseteq\sqsubseteq M_{\underline{G}} \quad (\underline{¿}\beta)$$

$$\bullet : \underline{¿} \vdash \bullet \sqsupseteq\sqsubseteq \{\& \mapsto \langle \underline{¿}\,\&\,\underline{¿} \swarrow \underline{¿}\rangle \bullet \mid (\to) \mapsto \langle ? \to \underline{¿} \swarrow \underline{¿}\rangle \bullet \mid \underline{F} \mapsto \langle \underline{F}? \swarrow \underline{¿}\rangle \bullet\} \quad (\underline{¿}\eta)$$

Figure 7.3: Natural Dynamic Type Extension of GTT

Then clearly by the roll/unroll isomorphism we get the desired isomorphisms:

$$? \cong ?_o[\underline{¿}/\underline{Y}, ?/X] = 1 + (? \times ?) + (? + ?) + U\underline{¿}$$
$$\underline{¿} \cong ?_c[?/X, \underline{¿}/\underline{Y}] = (\underline{¿}\,\&\,\underline{¿})\,\&\,(? \to \underline{¿})\,\&\,\underline{F}?$$

□

This dynamic type interpretation is a natural fit for CBPV because the introduction forms for ? are exactly the introduction forms for all of the value types (unit, pairing, `inl`, `inr`, `force`), while elimination forms are all of the elimination forms for computation types (π, π', application and binding); such "bityped" languages are related to Girard [30] and Zeilberger [92].

Based on this dynamic type interpretation, we can extend GTT to support a truly dynamically typed style of programming, where one can perform case-analysis on the dynamic types at runtime, in addition to the type assertions provided by upcasts and downcasts.

The axioms we choose might seem to under-specify the dynamic type, but because of the uniqueness of adjoints, the following are derivable.

Lemma 116 (Natural Dynamic Type Extension Theorems)**.** *The following are derivable in GTT with the natural dynamic type extension*

$$\langle \underline{F}1 \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_1.\mathtt{ret}\ x_1 \mid \mathtt{else}\ \mho\}$$

$$\langle \underline{F}(? \times ?) \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_\times.\mathtt{ret}\ x_\times \mid \mathtt{else}\ \mho\}$$

$$\langle \underline{F}(? + ?) \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_+.\mathtt{ret}\ x_+ \mid \mathtt{else}\ \mho\}$$

$$\langle \underline{F}U\underline{¿} \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_U.\mathtt{ret}\ x_U \mid \mathtt{else}\ \mho\}$$

$$\mathtt{force}\ \langle U\underline{¿} \nwarrow U(\underline{¿}\ \&\ \underline{¿}) \rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \mathtt{force}\ V \mid (\to) \mapsto \mho \mid \underline{F} \mapsto \mho\}$$

$$\mathtt{force}\ \langle U\underline{¿} \nwarrow U(? \to \underline{¿}) \rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \mho \mid (\to) \mapsto \mathtt{force}\ V \mid \underline{F} \mapsto \mho\}$$

$$\mathtt{force}\ \langle U\underline{¿} \nwarrow U\underline{F}? \rangle V \sqsupseteq\sqsubseteq \{\& \mapsto \mho \mid (\to) \mapsto \mho \mid \underline{F} \mapsto \mathtt{force}\ V\}$$

We explore this in more detail with the Scheme-like dynamic type interpretation below.

Next, we easily see that if we want to limit GTT to just the CBV types (i.e. the only computation types are $A \to \underline{F}A'$), then we can restrict the dynamic types as follows:

Definition 117 (CBV Dynamic Type Interpretation)**.** The following is a dynamic type interpretation for the ground types of GTT with only function computation types:

$$? \cong 1 + (? + ?) + (? \times ?) + U\underline{¿} \qquad \underline{¿} = ? \to \underline{F}?$$

And finally if we restrict GTT to only CBN types (i.e., the only value type is booleans $1 + 1$), we can restrict the dynamic types as follows:

Definition 118 (CBN Dynamic Type Interpretation)**.** The following is a dynamic type interpretation for the ground types of GTT with only boolean value types:

$$? = 1 + 1 \qquad \underline{¿} \cong (\underline{¿}\ \&\ \underline{¿})\ \&\ (U\underline{¿} \to \underline{¿})\ \&\ \underline{F}?$$

7.2.2 *Scheme-Like Dynamic Type Interpretation*

The above dynamic type interpretations do not correspond to any dynamically typed language used in practice, in part because it includes explicit cases for the "additives", the sum type $+$ and lazy product type $\&$. Normally, these are not included in this way, but rather sums are encoded by making each case use a fresh constructor (using nominal techniques like opaque structs in Racket) and then making the sum the union of the constructors, as argued in Siek and Tobin-Hochstadt [74]. We leave modeling this nominal structure to future work, possibly using the fresh type generation model of New,

Jamner, and Ahmed [57], but in minimalist languages, such as simple dialects of Scheme and Lisp, sum types are often encoded *structurally* rather than nominally by using some fixed sum type of *symbols*, also called *atoms*. Then a value of a sum type is modeled by a pair of a symbol (to indicate the case) and a payload with the actual value. We can model this by using the canonical isomorphisms

$$? + ? \cong ((1+1) \times ?) \qquad \underline{¿} \,\&\, \underline{¿} \cong (1+1) \to \underline{¿}$$

and representing sums as pairs, and lazy products as functions.

The fact that isomorphisms are ep pairs is useful for constructing the ep pairs needed in this Scheme-like dynamic type interpretation.

Lemma 119 (Isomorphisms are EP Pairs)**.** *If $x : A \vdash V' : A'$ and $x' : A' \vdash V : A$ are an isomorphism in that $V[V'/x'] \sqsupseteq\sqsubseteq x$ and $V[V/x] \sqsupseteq\sqsubseteq x'$, then $(x.V', x' \leftarrow \bullet; \texttt{ret}\ V')$ are a value ep pair from A to A'. Similarly if $\bullet : \underline{B} \vdash S' : \underline{B}'$ and $\bullet : \underline{B}' \vdash S : \underline{B}$ are an isomorphism in that $S[S'] \equiv \bullet$ and $S'[S] \equiv \bullet$ then $(z.S'[\texttt{force}\ z], S)$ is an ep pair from $\underline{B}$ to $\underline{B}'$.*

So we remove the cases for sums and lazy pairs from the natural dynamic types, and include some atomic type as a case of ?—for simplicity we will just use booleans. We also do not need a case for 1, because we can identify it with one of the booleans, say `true`. This leads to the following definition:

Definition 120 (Scheme-Like Dynamic Type Interpretation)**.** We can define a dynamic type interpretation with the following type isomorphisms:

$$? \cong (1+1) + U\underline{¿} + (? \times ?) \qquad\qquad \underline{¿} \cong (? \to \underline{¿}) \,\&\, \underline{F}?$$

Proof. We construct $?, \underline{¿}$ explicitly as follows.

First define $X :$ val type $\vdash \texttt{Tree}[X]$ val type to be the type of binary trees:

$$\texttt{Tree} = \mu X'.X + (X' \times X')$$

Next, define $X :$ val type, $\underline{Y} :$ comp type $\vdash \texttt{VarArg}[X, \underline{Y}]$ comp type to be the type of variable-arity functions from X to $\underline{Y}$:

$$\texttt{VarArg} = \nu \underline{Y}'.\underline{Y} \,\&\, (X \to \underline{Y}')$$

Then we define an open version of $?, \underline{¿}$ with respect to a variable representing the occurrences of ? in $\underline{¿}$:

$$X \text{ val type} \vdash ?_o = \texttt{Tree}[(1+1) + U\underline{¿}_o] \text{ val type}$$
$$X \text{ val type} \vdash \underline{¿}_o = \texttt{VarArg}[\underline{F}X/\underline{Y}] \text{ comp type}$$

Then we can define the closed versions using a recursive type:

$$? = \mu X.?_o \qquad\qquad \underline{¿} = \underline{¿}_o[?]$$

The ep pairs for $\times, U, \underline{F}, \to$ are clear. To define the rest, first note that there is an ep pair from $1+1$ to ? by Lemma 112. Next, we can define 1 to be the ep pair to $1+1$ defined by the left case and Lemma 112, composed with this. The ep pair for $?+?$ is defined by composing the isomorphism (which is always an ep pair) $(?+?) \cong ((1+1)\times ?)$ with the ep pair for $1+1$ using the action of product types on ep pairs (proven as part of Theorem 128): $(?+?) \cong ((1+1)\times ?) \triangleleft (?\times ?) \triangleleft ?$ (where we write $A \triangleleft A'$ to mean there is an ep pair from A to A'). Similarly, for $\underline{¿}\,\&\,\underline{¿}$, we use action of the function type on ep pairs (also proven as part of Theorem 128): $\underline{¿}\,\&\,\underline{¿} \cong ((1+1)\to \underline{¿}) \triangleleft (?\to \underline{¿}) \triangleleft \underline{¿}$ □

If we factor out some of the recursion to use inductive and coinductive types, we get the following isomorphisms:

$$? \cong \texttt{Tree}[(1+1)+U\underline{¿}/X] \qquad\qquad \underline{¿} \cong \texttt{VarArg}[?/X][\underline{F}?/\underline{Y}]$$

That is a dynamically typed value is a binary tree whose leaves are either booleans or closures. We think of this as a simple type of S-expressions. A dynamically typed computation is a variable-arity function that is called with some number of dynamically typed value arguments ? and returns a dynamically typed result $\underline{F}?$. This captures precisely the function type of Scheme, which allows for variable arity functions!

What's least clear is *why* the type

$$\texttt{VarArg}[X][\underline{Y}] = \nu\underline{Y}'.(X\to\underline{Y}')\,\&\,\underline{Y}$$

should be thought of as a type of variable arity functions. First consider the infinite unrolling of this type:

$$\texttt{VarArg}[X][\underline{Y}] \simeq \underline{Y}\,\&\,(X\to\underline{Y})\,\&\,(X\to X\to\underline{Y})\,\&\cdots$$

this says that a term of type $\texttt{VarArg}[X][Y]$ offers an infinite number of possible behaviors: it can act as a function from $X^n \to \underline{Y}$ for any n. Similarly in Scheme, a function can be called with any number of arguments. Finally note that this type is isomorphic to a function that takes a *cons-list* of arguments:

$$\begin{aligned}
&\underline{Y}\,\&\,(X\to\underline{Y})\,\&\,(X\to X\to\underline{Y})\,\&\cdots\\
&\cong (1\to\underline{Y})\,\&\,((X\times 1)\to\underline{Y})\,\&\,((X\times X\times 1)\to\underline{Y})\,\&\cdots\\
&\cong (1+(X\times 1)+(X\times X\times 1)+\cdots)\to\underline{Y}\\
&\cong (\mu X'.1+(X\times X'))\to\underline{Y}
\end{aligned}$$

But operationally the type $\texttt{VarArg}[?][\underline{F}?]$ is more faithful model of Scheme implementations that use the C-calling convention because all

of the arguments are passed individually on the stack, whereas the type $(\mu X.1 + (? \times X)) \to \underline{F}X$ is a function that takes a single argument that is a list. These two are distinguished in Scheme and the "dot args" notation witnesses the isomorphism.

Based on this dynamic type interpretation we can make a "Scheme-like" extension to GTT in Figure 7.4. First, we add a boolean type `Bool` with `true`, `false` and if-then-else. Next, we add in the elimination form for ? and the introduction form for ¿. The elimination form for ? is a typed version of Scheme's *match* macro. The introduction form for ¿ is a typed, CBPV version of Scheme's *case-lambda* construct. Finally, we add type precision rules expressing the representations of 1, $A + A$, and $A \times A$ in terms of booleans that were explicit in the ep pairs used in Definition 120.

The reader may be surprised by how *few* axioms we need to add to GTT for this extension: for instance we only define the upcast from 1 to `Bool` and not vice-versa, and similarly the sum/lazy pair type isomorphisms only have one cast defined when a priori there are 4 to be defined. Finally for the dynamic types we define β and η laws that use the ground casts as injections and projections respectively, but we don't define the corresponding dual casts (the ones that possibly error).

In fact all of these expected axioms can be *proven* from those we have shown. Again we see the surprising rigidity of GTT: because an $\underline{F}$ downcast is determined by its dual value upcast (and vice-versa for U upcasts), we only need to define the upcast as long as the downcast *could* be implemented already. Because we give the dynamic types the universal property of a sum/lazy product type respectively, we can derive the implementations of the "checking" casts. All of the proofs are direct from the uniqueness of adjoints lemma.

Theorem 121 (Boolean to Unit Downcast)**.** *In Scheme-like GTT, we can prove*

$$\langle \underline{F}1 \swarrow \underline{F}\texttt{Bool} \rangle \bullet \sqsupseteq\sqsubseteq x \leftarrow \bullet; \texttt{if } x \texttt{ then ret } () \texttt{ else } \Omega$$

Theorem 122 (Tagged Value to Sum)**.** *In Scheme-like GTT, we can prove*

$$\langle A + A \nwarrow \texttt{Bool} \times A \rangle V \sqsupseteq\sqsubseteq \texttt{let } (x,y) = V; \texttt{if } x \texttt{ then inl } y \texttt{ else inr } y$$

and the downcasts are given by Lemma 119.

Theorem 123 (Lazy Product to Tag Checking Function)**.** *In Scheme-like GTT, we can prove*

$$\langle \texttt{Bool} \to \underline{B} \swarrow \underline{B} \,\&\, \underline{B} \rangle \bullet \sqsupseteq\sqsubseteq \lambda x : \texttt{Bool}. \texttt{if } x \texttt{ then } \pi \bullet \texttt{ else } \pi' \bullet$$

and the upcasts are given by Lemma 119.

$$1 \sqsubseteq \mathtt{Bool} \qquad A + A \sqsupseteq\sqsubseteq \mathtt{Bool} \times A \qquad \underline{B} \,\&\, \underline{B} \sqsupseteq\sqsubseteq \mathtt{Bool} \to \underline{B}$$

$$\frac{}{\Gamma \vdash \mathtt{true}, \mathtt{false} : \mathtt{Bool}}\ \mathtt{Bool}\mathrm{I}$$

$$\frac{\Gamma \vdash V : \mathtt{Bool} \qquad \Gamma \vdash E_t : T \qquad \Gamma \vdash E_f : T}{\Gamma \mid \Delta \vdash \mathtt{if}\ V\ \mathtt{then}\ E_t\ \mathtt{else}\ E_f : T}\ \mathtt{Bool}\mathrm{E}$$

$$\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ E_t\ \mathtt{else}\ E_f \sqsupseteq\sqsubseteq E_t \qquad \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ E_t\ \mathtt{else}\ E_f \sqsupseteq\sqsubseteq E_f$$

$$x : \mathtt{Bool} \vdash E \sqsupseteq\sqsubseteq \mathtt{if}\ x\ \mathtt{then}\ E[\mathtt{true}/x]\ \mathtt{else}\ E[\mathtt{false}/x]$$

$$\langle \mathtt{Bool} \nwarrow 1 \rangle V \sqsupseteq\sqsubseteq \mathtt{true} \qquad \langle \mathtt{Bool} \times A \nwarrow A + A \rangle \mathtt{inl}\ V \sqsupseteq\sqsubseteq (\mathtt{true}, V)$$

$$\langle \mathtt{Bool} \times A \nwarrow A + A \rangle \mathtt{inr}\ V \sqsupseteq\sqsubseteq (\mathtt{false}, V)$$

$$\pi \langle \underline{B} \,\&\, \underline{B} \swarrow \mathtt{Bool} \to \underline{B} \rangle M \sqsupseteq\sqsubseteq M\,\mathtt{true}$$

$$\pi' \langle \underline{B} \,\&\, \underline{B} \swarrow \mathtt{Bool} \to \underline{B} \rangle M \sqsupseteq\sqsubseteq M\,\mathtt{false}$$

$$\frac{\Gamma \mid \Delta \vdash M_{\to} : ? \to \underline{¿} \qquad \Gamma \mid \Delta \vdash M_{\underline{F}} : \underline{F}?}{\Gamma \mid \Delta \vdash \{(\to) \mapsto M_{\to} \mid \underline{F} \mapsto M_{\underline{F}}\} : \underline{¿}}\ \underline{¿}\mathrm{I}$$

$$\langle \underline{G} \swarrow \underline{¿} \rangle \{(\to) \mapsto M_{\to} \mid \underline{F} \mapsto M_{\underline{F}}\} \sqsupseteq\sqsubseteq M_{\underline{G}} \quad (\underline{¿}\beta)$$

$$\bullet : \underline{¿} \vdash \bullet \sqsupseteq\sqsubseteq \{(\to) \mapsto \langle ? \to \underline{¿} \swarrow \underline{¿} \rangle \bullet \mid \underline{F} \mapsto \langle \underline{F}? \swarrow \underline{¿} \rangle \bullet\} \quad (\underline{¿}\eta)$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta \vdash V : ? \\ \Gamma, x_{\mathtt{Bool}} : \mathtt{Bool} \mid \Delta \vdash E_{\mathtt{Bool}} : T \\ \Gamma, x_U : U_{\underline{¿}} \mid \Delta \vdash E_U : T \\ \Gamma, x_\times : ? \times ? \mid \Delta \vdash E_\times : T\end{array}}{\Gamma \mid \Delta \vdash \mathtt{tycase}\ V\ \{x_{\mathtt{Bool}}.E_{\mathtt{Bool}} \mid x_U.E_U \mid x_\times.E_\times\} : T}\ ?\mathrm{E}$$

$$\frac{G \in \{\mathtt{Bool}, \times, U\}}{\mathtt{tycase}\ (\langle ? \nwarrow G \rangle V)\ \{x_{\mathtt{Bool}}.E_{\mathtt{Bool}} \mid x_U.E_U \mid x_\times.E_\times\} \sqsupseteq\sqsubseteq E_G[V/x_G]} \qquad (?\beta)$$

$$\frac{\Gamma, x : ? \mid \Delta \vdash E : \underline{B}}{E \sqsupseteq\sqsubseteq \mathtt{tycase}\ x\ \{x_{\mathtt{Bool}}.E[\langle ? \nwarrow \mathtt{Bool} \rangle / x_{\mathtt{Bool}}] \mid x_\times.E[\langle ? \nwarrow \times \rangle / x_\times] \mid x_U.E[\langle ? \nwarrow U \rangle / x_U]\}}\ ?\eta$$

Figure 7.4: Scheme-like Extension to GTT

Theorem 124 (Ground Mismatches are Errors)**.** *In Scheme-like GTT we can prove*

$$\langle \underline{F}\mathtt{Bool} \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_{\mathtt{Bool}}.\mathtt{ret}\ x_{\mathtt{Bool}} \mid \mathtt{else}\ \mho\}$$

$$\langle \underline{F}(? \times ?) \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_{\times}.\mathtt{ret}\ x_{\times} \mid \mathtt{else}\ \mho\}$$

$$\langle \underline{F}U\underline{¿} \swarrow \underline{F}? \rangle \mathtt{ret}\ V \sqsupseteq\sqsubseteq \mathtt{tycase}\ V\ \{x_{U}.\mathtt{ret}\ x_{U} \mid \mathtt{else}\ \mho\}$$

$$\mathtt{force}\ \langle U\underline{¿} \nwarrow U(? \to \underline{¿}) \rangle V \sqsupseteq\sqsubseteq \{(\to) \mapsto \mathtt{force}\ V \mid \underline{F} \mapsto \mho\}$$

$$\mathtt{force}\ \langle U\underline{¿} \nwarrow U\underline{F}? \rangle V \sqsupseteq\sqsubseteq \{(\to) \mapsto \mho \mid \underline{F} \mapsto \mathtt{force}\ V\}$$

Next, note that this model gives an example of why the disjointness of type constructors we encode into different calculi in Chapter 6 can be derived from GTT. In the call-by-value calculus, any cast from a sum type to a product type would fail, but here we have a model where all sum types can be safely cast to $\mathtt{Bool} \times ?$.

Finally, we note now that all of these axioms are satisfied when using the Scheme-like dynamic type interpretation and extending the translation of GTT into CBPV* with the following, tediously explicit definition:

$$\begin{aligned}
&[\![\mathtt{Bool}]\!] = 1 + 1\\
&[\![\mathtt{true}]\!] = \mathtt{inl}\ ()\\
&[\![\mathtt{false}]\!] = \mathtt{inr}\ ()\\
&[\![\mathtt{if}\ V\ \mathtt{then}\ E_t\ \mathtt{else}\ E_f]\!] = \mathtt{case}\ [\![V]\!]\{x.E_t \mid x.E_f\}\\
&[\![\mathtt{tycase}\ x\ \{x_{\mathtt{Bool}}.E_{\mathtt{Bool}} \mid x_U.E_U \mid x_\times.E_\times\}]\!] =\\
&\quad \mathtt{unroll}\ (x : ?)\ \mathtt{to}\ \mathtt{roll}\ x'.\mathtt{unroll}\ x' : \mathtt{Tree}[(1+1) + U\underline{¿}]\ \mathtt{to}\ \mathtt{roll}\ t.\mathtt{case}\ t\\
&\qquad \{l.\mathtt{case}\ l\{x_{\mathtt{Bool}}.[\![E_{\mathtt{Bool}}]\!] \mid x_U.[\![E_U]\!]\}\\
&\qquad\ \mid x_\times.[\![E_\times]\!]\}\\
&[\![\{(\to) \mapsto M_\to \mid \underline{F} \mapsto M_{\underline{F}}\}]\!] = \mathtt{roll}_{\nu \underline{Y}.(? \to \underline{Y}) \& \underline{F}?}\ ([\![M_\to]\!], [\![M_{\underline{F}}]\!])
\end{aligned}$$

7.2.3 *Contract Translation*

Having defined the data parameterizing the translation, we now consider the translation of GTT into CBPV* itself. For the remainder of the paper, we assume that we have a fixed dynamic type interpretation ρ, and all proofs and definitions work for any interpretation.

7.2.3.1 *Interpreting Casts as Contracts*

The main idea of the translation is an extension of the dynamic type interpretation to an interpretation of *all* casts in GTT (Figure 7.5) as contracts in CBPV*, following the definitions in Lemma 81. Some

$$x : [\![A]\!] \vdash [\![\langle A' \nwarrow A\rangle]\!] : [\![A']\!] \qquad \bullet : [\![\underline{B}']\!] \vdash [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!] : [\![\underline{B}]\!]$$

$$\begin{array}{rcl}
x : 0 \vdash [\![\langle A \nwarrow 0\rangle]\!] & = & \mathsf{absurd}\ x \\
\bullet : A \vdash [\![\langle \underline{F}0 \swarrow \underline{F}A\rangle]\!] & = & x \leftarrow \bullet; \mho \\
x : [\![?]\!] \vdash [\![\langle ? \nwarrow ?\rangle]\!] & = & x \\
\bullet : \underline{F}? \vdash [\![\langle \underline{F}? \swarrow \underline{F}?\rangle]\!] & = & \bullet \\
x : [\![G]\!] \vdash [\![\langle ? \nwarrow G\rangle]\!] & = & \rho_{up}(G) \\
\bullet : \underline{F}? \vdash [\![\langle \underline{F}G \swarrow \underline{F}?\rangle]\!] & = & \rho_{dn}(G) \\
x : [\![A]\!] \vdash [\![\langle ? \nwarrow A\rangle]\!] & = & [\![\langle ? \nwarrow \lfloor A\rfloor\rangle]\!][[\![\langle \lfloor A\rfloor \nwarrow A\rangle]\!]/x] \\
\bullet : \underline{F}? \vdash [\![\langle A \swarrow ?\rangle]\!] & = & [\![\langle A \swarrow \lfloor A\rfloor\rangle]\!][[\![\langle \lfloor A\rfloor \swarrow ?\rangle]\!]] \\
x \vdash [\![\langle A'_1 + A'_2 \nwarrow A_1 + A_2\rangle]\!] & = & \mathsf{case}\ x \\
 & & \{x_1.[\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1/x] \\
 & & \mid x_2.[\![\langle A'_2 \nwarrow A_2\rangle]\!][x_2/x]\} \\
\bullet \vdash [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A'_1 + A'_2)\rangle]\!] & = & x' \leftarrow \bullet; \mathsf{case}\ x' \\
 & & \{x'_1.x_1 \leftarrow ([\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!]\mathsf{ret}\ x'_1); \mathsf{ret}\ x_1 \\
 & & \mid x'_2.x_2 \leftarrow ([\![\langle \underline{F}A_2 \swarrow \underline{F}A'_2\rangle]\!]\mathsf{ret}\ x'_2); \mathsf{ret}\ x_2\} \\
x : 1 \vdash [\![\langle 1 \nwarrow 1\rangle]\!] & = & x \\
\bullet : \underline{F}1 \vdash [\![\langle \underline{F}1 \swarrow \underline{F}1\rangle]\!] & = & x \\
x \vdash [\![\langle A'_1 \times A'_2 \nwarrow A_1 \times A_2\rangle]\!] & = & \mathsf{let}\ (x_1, x_2) = x; \\
 & & ([\![\langle A'_1 \nwarrow A_1\rangle]\!][x_1], [\![\langle A'_2 \nwarrow A_2\rangle]\!][x_2]) \\
\bullet \vdash [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A'_1 \times A'_2)\rangle]\!] & = & x' \leftarrow \bullet; \mathsf{let}\ (x'_1, x'_2) = x'; \\
 & & x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A'_1\rangle]\!]\mathsf{ret}\ x'_1; \\
 & & x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A'_2\rangle]\!]\mathsf{ret}\ x'_2; \mathsf{ret}\ (x_1, x_2) \\
x : U\underline{F}[\![A]\!] \vdash [\![\langle U\underline{F}A' \nwarrow U\underline{F}A\rangle]\!] & = & \mathsf{thunk}\ (y \leftarrow \mathsf{force}\ x; \mathsf{ret}\ [\![\langle A' \nwarrow A\rangle]\!][y/x]) \\
 & & \\
\bullet : \underline{B} \vdash [\![\langle \top \swarrow \underline{B}\rangle]\!] & = & \{\} \\
x : U\top \vdash [\![\langle U\underline{B} \nwarrow U\top\rangle]\!] & = & \mathsf{thunk}\ \mho \\
\bullet : \underline{\iota} \vdash [\![\langle \underline{\iota} \swarrow \underline{\iota}\rangle]\!] & = & \bullet \\
x : U\underline{\iota} \vdash [\![\langle U\underline{\iota} \nwarrow U\underline{\iota}\rangle]\!] & = & x \\
\bullet : \underline{\iota} \vdash [\![\langle \underline{G} \swarrow \underline{\iota}\rangle]\!] & = & \rho_{dn}(\underline{G}) \\
x : U\underline{G} \vdash [\![\langle U\underline{\iota} \nwarrow U\underline{G}\rangle]\!] & = & \rho_{up}(\underline{G}) \\
\bullet : \underline{\iota} \vdash [\![\langle \underline{B} \swarrow \underline{\iota}\rangle]\!] & = & [\![\langle \underline{B} \swarrow \lfloor \underline{B}\rfloor\rangle]\!][[\![\langle \lfloor \underline{B}\rfloor \swarrow \underline{\iota}\rangle]\!]] \\
x : U\underline{\iota} \vdash [\![\langle U\underline{\iota} \nwarrow U\underline{B}\rangle]\!] & = & [\![\langle U\underline{\iota} \nwarrow U\lfloor \underline{B}\rfloor\rangle]\!][[\![\langle U\lfloor \underline{B}\rfloor \nwarrow U\underline{B}\rangle]\!]] \\
\bullet \vdash [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}'_1 \,\&\, \underline{B}'_2\rangle]\!] & = & \{\pi \mapsto [\![\langle \underline{B}_1 \swarrow \underline{B}'_1\rangle]\!]\pi\bullet \\
 & & \mid \pi' \mapsto [\![\langle \underline{B}_2 \swarrow \underline{B}'_2\rangle]\!]\pi'\bullet\} \\
x \vdash [\![\langle U(\underline{B}'_1 \,\&\, \underline{B}'_2) \nwarrow U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!] & = & \mathsf{thunk} \\
 & & \{\pi \mapsto \mathsf{force}\ [\![\langle \underline{B}'_1 \nwarrow \underline{B}_1\rangle]\!](\mathsf{thunk}\ \pi\mathsf{force}\ x) \\
 & & \mid \pi' \mapsto \mathsf{force}\ [\![\langle \underline{B}'_2 \nwarrow \underline{B}_2\rangle]\!](\mathsf{thunk}\ \pi'\mathsf{force}\ x)\} \\
\bullet \vdash [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!] & = & \lambda x : A.[\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!](\bullet\,([\![\langle A' \nwarrow A\rangle]\!]x)) \\
f \vdash [\![\langle U(A' \to \underline{B}') \nwarrow U(A \to \underline{B})\rangle]\!] & = & \mathsf{thunk}\ \lambda x' : A'. \\
 & & x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!]\mathsf{ret}\ x'; \\
 & & \mathsf{force}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!]\mathsf{thunk}\ (\mathsf{force}\ f)\ x' \\
\bullet : \underline{F}U\underline{B}' \vdash [\![\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle]\!] & = & x' \leftarrow \bullet; [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!]\mathsf{force}\ x'
\end{array}$$

Figure 7.5: Cast to Contract Translation

$$\frac{A \in \{?, 1\}}{A \sqsubseteq A} \qquad \frac{A \in \{?, 0\}}{0 \sqsubseteq A} \qquad \frac{A \sqsubseteq \lfloor A \rfloor \qquad A \notin \{0, ?\}}{A \sqsubseteq ?}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}'}{U\underline{B} \sqsubseteq U\underline{B}'} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 + A_2 \sqsubseteq A_1' + A_2'} \qquad \frac{A_1 \sqsubseteq A_1' \qquad A_2 \sqsubseteq A_2'}{A_1 \times A_2 \sqsubseteq A_1' \times A_2'}$$

$$\underline{¿} \sqsubseteq \underline{¿} \qquad \frac{\underline{B} \in \{\underline{¿}, \top\}}{\top \sqsubseteq \underline{B}} \qquad \frac{\underline{B} \sqsubseteq \lfloor \underline{B} \rfloor \qquad \underline{B} \notin \{\top, \underline{¿}\}}{\underline{B} \sqsubseteq \underline{¿}}$$

$$\frac{A \sqsubseteq A'}{\underline{F}A \sqsubseteq \underline{F}A'} \qquad \frac{\underline{B}_1 \sqsubseteq \underline{B}_1' \qquad \underline{B}_2 \sqsubseteq \underline{B}_2'}{\underline{B}_1 \,\&\, \underline{B}_2 \sqsubseteq \underline{B}_1' \,\&\, \underline{B}_2'} \qquad \frac{A \sqsubseteq A' \qquad \underline{B} \sqsubseteq \underline{B}'}{A \to \underline{B} \sqsubseteq A' \to \underline{B}'}$$

Figure 7.6: Normalized Type Precision Relation

clauses of the translation are overlapping, which we resolve by considering them as ordered (though we will ultimately show they are equivalent). The definition is also not obviously total: we need to verify that it covers every possible case where $A \sqsubseteq A'$ and $\underline{B} \sqsubseteq \underline{B}'$. To prove totality and coherence, we could try induction on the type precision relation of Figure 5.3, but it is convenient to first give an alternative, normalized set of rules for type precision that proves the same relations, which we do in Figure 7.6.

Lemma 125 (Normalized Type Precision is Equivalent to Original)**.** *$T \sqsubseteq T'$ is provable in the normalized typed precision definition iff it is provable in the original typed precision definition.*

Proof. It is clear that the normalized system is a subset of the original: every normalized rule corresponds directly to a rule of the original system, except the normalized $A \sqsubseteq ?$ and $\underline{B} \sqsubseteq \underline{¿}$ rules have a subderivation that was not present originally.

For the converse, first we show by induction that reflexivity is admissible:

1. If $A \in \{?, 1, 0\}$, we use a normalized rule.
2. If $A \notin \{?, 1, 0\}$, we use the inductive hypothesis and the monotonicity rule.
3. If $\underline{B} \in \{\underline{¿}, \top\}$ use the normalized rule.
4. If $\underline{B} \notin \{\underline{¿}, \top\}$ use the inductive hypothesis and monotonicity rule.

Next, we show that transitivity is admissible:

1. Assume we have $A \sqsubseteq A' \sqsubseteq A''$

a) If the left rule is $0 \sqsubseteq A'$, then either $A' = ?$ or $A' = 0$. If $A' = 0$ the right rule is $0 \sqsubseteq A''$ and we can use that proof. Otherwise, $A' = ?$ then the right rule is $? \sqsubseteq ?$ and we can use $0 \sqsubseteq ?$.

b) If the left rule is $A \sqsubseteq A$ where $A \in \{?, 1\}$ then either $A = ?$, in which case $A'' = ?$ and we're done. Otherwise the right rule is either $1 \sqsubseteq 1$ (done) or $1 \sqsubseteq ?$ (also done).

c) If the left rule is $A \sqsubseteq ?$ with $A \notin \{0, ?\}$ then the right rule must be $? \sqsubseteq ?$ and we're done.

d) Otherwise the left rule is a monotonicity rule for one of $U, +, \times$ and the right rule is either monotonicity (use the inductive hypothesis) or the right rule is $A' \sqsubseteq ?$ with a sub-proof of $A' \sqsubseteq \lfloor A' \rfloor$. Since the left rule is monotonicity, $\lfloor A \rfloor = \lfloor A' \rfloor$, so we inductively use transitivity of the proof of $A \sqsubseteq A'$ with the proof of $A' \sqsubseteq \lfloor A' \rfloor$ to get a proof $A \sqsubseteq \lfloor A \rfloor$ and thus $A \sqsubseteq ?$.

2. Assume we have $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$.

a) If the left rule is $\top \sqsubseteq \underline{B}'$ then $\underline{B}'' \in \{\underline{¿}, \top\}$ so we apply that rule.

b) If the left rule is $\underline{¿} \sqsubseteq \underline{¿}$, the right rule must be as well.

c) If the left rule is $\underline{B} \sqsubseteq \underline{¿}$ the right rule must be reflexivity.

d) If the left rule is a monotonicity rule for $\&, \to, \underline{F}$ then the right rule is either also monotonicity (use the inductive hypothesis) or it's a $\underline{B} \sqsubseteq \underline{¿}$ rule and we proceed with ? above

Finally we show $A \sqsubseteq ?$, $\underline{B} \sqsubseteq \underline{¿}$ are admissible by induction on A, $\underline{B}$.

1. If $A \in \{?, 0\}$ we use the primitive rule.

2. If $A \notin \{?, 0\}$ we use the $A \sqsubseteq ?$ rule and we need to show $A \sqsubseteq \lfloor A \rfloor$. If $A = 1$, we use the $1 \sqsubseteq 1$ rule, otherwise we use the inductive hypothesis and monotonicity.

3. If $\underline{B} \in \{\underline{¿}, \top\}$ we use the primitive rule.

4. If $\underline{B} \notin \{\underline{¿}, \top\}$ we use the $\underline{B} \sqsubseteq \underline{¿}$ rule and we need to show $\underline{B} \sqsubseteq \lfloor \underline{B} \rfloor$, which follows by inductive hypothesis and monotonicity.

Every other rule in Figure 5.3 is a rule of the normalized system in Figure 7.6. □

Based on normalized type precision, we show

Theorem 126. *If $A \sqsubseteq A'$ according to Figure 7.6, then there is a unique complex value $x : A \vdash [\![\langle A' \nwarrow A \rangle]\!] x : A'$ and if $\underline{B} \sqsubseteq \underline{B}'$ according to Figure 7.6, then there is a unique complex stack $x : \underline{B} \vdash [\![\langle \underline{B}' \nwarrow \underline{B} \rangle]\!] x : \underline{B}'$*

7.2.3.2 *Interpretation of Terms*

Next, we extend the translation of casts to a translation of all terms by congruence, since all terms in GTT besides casts are in CBPV*. This satisfies:

Lemma 127 (Contract Translation Type Preservation)**.** *If* $\Gamma \mid \Delta \vdash E : T$ *in GTT, then* $[\![\Gamma]\!] \mid [\![\Delta]\!] \vdash [\![E]\!] : [\![T]\!]$ *in CBPV*.*

7.2.3.3 *Interpretation of Term Precision*

We have now given an interpretation of the types, terms, and type precision proofs of GTT in CBPV*. To complete this to form a *model* of GTT, we need to give an interpretation of the *term precision* proofs, which is established by the following "axiomatic graduality" theorem. GTT has *heterogeneous* term precision rules indexed by type precision, but CBPV* has only *homogeneous* inequalities between terms, i.e., if $E \sqsubseteq E'$, then E, E' have the *same* context and types. Since every type precision judgement has an associated contract, we can translate a heterogeneous term precision to a homogeneous inequality *up to contract*. Our next overall goal is to prove our axiomatic graduality theorem:

Theorem 128 (Axiomatic Graduality)**.** *For any dynamic type interpretation,*

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Psi : \Delta \sqsubseteq \Delta' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{[\![\Gamma]\!] \mid [\![\Delta']\!] \vdash [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]}$$

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{[\![\Gamma]\!] \vdash [\![\langle A' \nwarrow A \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] : [\![A']\!]}$$

where we define $[\![\Phi]\!]$ *to upcast each variable, and* $[\![\Delta]\!]$ *to downcast* $\bullet$ *if it is nonempty, and if* $\Delta = \cdot$*, then* $M[[\![\Delta]\!]] = M$*. More explicitly,*

1. *If* $\Phi : \Gamma \sqsubseteq \Gamma'$*, then there exists* n *such that* $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ *and* $\Gamma' = x'_1 : A'_1, \ldots, x'_n : A'_n$ *where* $A_i \sqsubseteq A'_i$ *for each* $i \leq n$*. Then* $[\![\Phi]\!]$ *is a substitution from* $[\![\Gamma]\!]$ *to* $[\![\Gamma']\!]$ *defined as*

$$[\![\Phi]\!] = [\![\langle A'_1 \nwarrow A_1 \rangle]\!] x_1 / x'_1, \ldots [\![\langle A'_n \nwarrow A_n \rangle]\!] x_n / x'_n$$

2. *If* $\Psi : \Delta \sqsubseteq \Delta'$*, then we similarly define* $[\![\Psi]\!]$ *as a "linear substitution". That is, if* $\Delta = \Delta' = \cdot$*, then* $[\![\Psi]\!]$ *is an empty substitution and* $M[[\![\Psi]\!]] = M$*, otherwise* $[\![\Psi]\!]$ *is a linear substitution from* $\Delta' = \bullet : \underline{B}'$ *to* $\Delta = \bullet : \underline{B}$ *where* $\underline{B} \sqsubseteq \underline{B}'$ *defined as*

$$[\![\Psi]\!] = [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!] \bullet / \bullet$$

Relative to previous work on graduality [56], the distinction between complex value upcasts and complex stack downcasts here guides the

formulation of the theorem; e.g. using upcasts in the left-hand theorem would require more thunks/forces.

We now develop some lemmas on the way towards proving this result. First, we prove that from the basic casts being ep pairs, we can prove that all casts as defined in Figure 7.5 are ep pairs. Before doing so, we prove the following lemma, which is used for transitivity (e.g. in the $A \sqsubseteq ?$ rule, which uses a composition $A \sqsubseteq \lfloor A \rfloor \sqsubseteq ?$):

Lemma 129 (EP Pairs Compose)**.**

1. *If (V_1, S_1) is a value ep pair from A_1 to A_2 and (V_2, S_2) is a value ep pair from A_2 to A_3, then $(V_2[V_1], S_1[S_2])$ is a value ep pair from A_1 to A_3.*
2. *If (V_1, S_1) is a computation ep pair from $\underline{B}_1$ to $\underline{B}_2$ and (V_2, S_2) is a computation ep pair from $\underline{B}_2$ to $\underline{B}_3$, then $(V_2[V_1], S_1[S_2])$ is a computation ep pair from $\underline{B}_1$ to $\underline{B}_3$.*

Proof. 1. First, retraction follows from retraction twice:

$$S_1[S_2[\mathtt{ret}\ V_2[V_1[x]]]] \sqsupseteq\sqsubseteq S_1[\mathtt{ret}\ [V_1[x]]] \sqsupseteq\sqsubseteq x$$

and projection follows from projection twice:

$$\begin{aligned} x \leftarrow S_1[S_2[\bullet]]; \mathtt{ret}\ V_2[V_1[x]] &\sqsupseteq\sqsubseteq x \leftarrow S_1[S_2[\bullet]]; y \leftarrow \mathtt{ret}\ [V_1[x]]; \mathtt{ret}\ V_2[y] && (\underline{F}\beta) \\ &\sqsupseteq\sqsubseteq y \leftarrow (x \leftarrow S_1[S_2[\bullet]]; \mathtt{ret}\ [V_1[x]]); \mathtt{ret}\ V_2[y] && \text{(Commuting conversion)} \\ &\sqsubseteq y \leftarrow S_2[\bullet]; \mathtt{ret}\ V_2[y] && \text{(Projection)} \\ &\sqsubseteq \bullet && \text{(Projection)} \end{aligned}$$

2. Again retraction follows from retraction twice:

$$S_1[S_2[\mathtt{force}\ V_2[V_1[z]]]] \sqsupseteq\sqsubseteq S_1[\mathtt{force}\ V_1[z]] \sqsupseteq\sqsubseteq \mathtt{force}\ z$$

and projection from projection twice:

$$\begin{aligned} V_2[V_1[\mathtt{thunk}\ S_1[S_2[\mathtt{force}\ w]]]] &\sqsupseteq\sqsubseteq V_2[V_1[\mathtt{thunk}\ S_1[\mathtt{force}\ \mathtt{thunk}\ S_2[\mathtt{force}\ w]]]] && (U\beta) \\ &\sqsubseteq V_2[\mathtt{thunk}\ S_2[\mathtt{force}\ w]] && \text{(Projection)} \\ &\sqsubseteq w && \text{(Projection)} \end{aligned}$$

□

Lemma 130 (Identity EP Pair)**.** *$(x.x, \bullet)$ is an ep pair (value or computation).*

Now, we show that all casts are ep pairs. The proof is a somewhat tedious, but straightforward calculation. To keep proofs high-level, we first establish the following cast reductions that follow easily from β, η principles.

Lemma 131 (Cast Reductions)**.** *The following are all provable*

$$[\![\langle A_1' + A_2' \leftharpoondown A_1 + A_2\rangle]\!][\texttt{inl}\ V] \sqsupseteq\sqsubseteq \texttt{inl}\ [\![\langle A_1' \leftharpoondown A_1\rangle]\!][V]$$
$$[\![\langle A_1' + A_2' \leftharpoondown A_1 + A_2\rangle]\!][\texttt{inr}\ V] \sqsupseteq\sqsubseteq \texttt{inr}\ [\![\langle A_2' \leftharpoondown A_2\rangle]\!][V]$$
$$[\![\langle \underline{F}(A_1 + A_2) \leftarrow \underline{F}(A_1' + A_2')\rangle]\!][\texttt{ret inl}\ V] \sqsupseteq\sqsubseteq x_1 \leftarrow [\![\langle A_1 \leftarrow A_1'\rangle]\!][\texttt{ret}\ V]; \texttt{ret inl}\ x_1$$
$$[\![\langle \underline{F}(A_1 + A_2) \leftarrow \underline{F}(A_1' + A_2')\rangle]\!][\texttt{ret inr}\ V] \sqsupseteq\sqsubseteq x_2 \leftarrow [\![\langle A_2 \leftarrow A_2'\rangle]\!][\texttt{ret}\ V]; \texttt{ret inr}\ x_2$$

$$[\![\langle \underline{F}1 \leftarrow \underline{F}1\rangle]\!] \sqsupseteq\sqsubseteq \bullet$$
$$[\![\langle 1 \leftharpoondown 1\rangle]\!][x] \sqsupseteq\sqsubseteq x$$

$$[\![\langle \underline{F}(A_1 \times A_2) \leftarrow \underline{F}(A_1' \times A_2')\rangle]\!][\texttt{ret}\ (V_1, V_2)]$$
$$\sqsupseteq\sqsubseteq x_1 \leftarrow [\![\langle \underline{F}A_1 \leftarrow \underline{F}A_1'\rangle]\!][\texttt{ret}\ V_1]; x_2 \leftarrow [\![\langle \underline{F}A_2 \leftarrow \underline{F}A_2'\rangle]\!][\texttt{ret}\ V_2]; \texttt{ret}\ (x_1, x_2)$$
$$[\![\langle A_1' \times A_2' \leftharpoondown A_1 \times A_2\rangle]\!][(V_1, V_2)] \sqsupseteq\sqsubseteq ([\![\langle A_1' \leftharpoondown A_1\rangle]\!][V_1], [\![\langle A_2' \leftharpoondown A_2\rangle]\!][V_2])$$

$$([\![\langle A \to \underline{B} \leftarrow A' \to \underline{B}'\rangle]\!]M)\ V \sqsupseteq\sqsubseteq ([\![\langle \underline{B} \leftarrow \underline{B}'\rangle]\!]M)\ ([\![\langle A' \leftharpoondown A\rangle]\!]V)$$
$$(\texttt{force}\ ([\![\langle U(A' \to \underline{B}') \leftharpoondown U(A \to \underline{B})\rangle]\!]V))\ V'$$
$$\sqsupseteq\sqsubseteq x \leftarrow \langle \underline{F}A \leftarrow \underline{F}A'\rangle[\texttt{ret}\ V']; \texttt{force}\ ([\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!](\texttt{thunk}\ (\texttt{force}\ V\ x)))$$

$$\pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \leftarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!]M \sqsupseteq\sqsubseteq [\![\langle \underline{B}_1 \leftarrow \underline{B}_1'\rangle]\!]\pi M$$
$$\pi'[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \leftarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!]M \sqsupseteq\sqsubseteq [\![\langle \underline{B}_2 \leftarrow \underline{B}_2'\rangle]\!]\pi' M$$
$$\pi\texttt{force}\ ([\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoondown U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!]V) \sqsupseteq\sqsubseteq \texttt{force}\ [\![\langle U\underline{B}_1' \leftharpoondown U\underline{B}_1\rangle]\!]\texttt{thunk}\ (\pi\texttt{force}\ V)$$
$$\pi'\texttt{force}\ ([\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoondown U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!]V) \sqsupseteq\sqsubseteq \texttt{force}\ [\![\langle U\underline{B}_2' \leftharpoondown U\underline{B}_2\rangle]\!]\texttt{thunk}\ (\pi'\texttt{force}\ V)$$

$$[\![\langle \underline{F}U\underline{B} \leftarrow \underline{F}U\underline{B}'\rangle]\!][\texttt{ret}\ V] \sqsupseteq\sqsubseteq \texttt{ret thunk}\ [\![\langle \underline{B} \leftarrow \underline{B}'\rangle]\!]\texttt{force}\ V$$
$$\texttt{force}\ [\![\langle U\underline{F}A' \leftharpoondown U\underline{F}A\rangle]\!][V] \sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ V; \texttt{thunk ret}\ \langle A' \leftharpoondown A\rangle x$$

Lemma 132 (Casts are EP Pairs)**.**

1. *For any $A \sqsubseteq A'$, the casts $(x.[\![\langle A' \leftharpoondown A\rangle x]\!], [\![\langle \underline{F}A \leftarrow \underline{F}A'\rangle]\!])$ are a value ep pair from $[\![A]\!]$ to $[\![A']\!]$*
2. *For any $\underline{B} \sqsubseteq \underline{B}'$, the casts $(z.[\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle z]\!], [\![\langle \underline{B} \leftarrow \underline{B}'\rangle]\!])$ are a computation ep pair from $[\![\underline{B}]\!]$ to $[\![\underline{B}']\!]$.*

Proof. By induction on normalized type precision derivations.

1. $A \sqsubseteq A$ ($A \in \{?, 1\}$), because identity is an ep pair.
2. $0 \sqsubseteq A$ (that $A \in \{?, 0\}$ is not important):
   a) Retraction is

$$x : 0 \vdash \texttt{ret}\ x \sqsupseteq\sqsubseteq y \leftarrow \texttt{ret absurd}\ x; \mho : \underline{F}A$$

   which holds by 0η

b) Projection is

$$\bullet : \underline{F}A \vdash x \leftarrow (y \leftarrow \bullet; \mho); \mathtt{ret}\ \mathtt{absurd}\ x \sqsubseteq \bullet : \underline{F}A$$

Which we calculate:

$$\begin{aligned}
& x \leftarrow (y \leftarrow \bullet; \mho); \mathtt{ret}\ \mathtt{absurd}\ x \\
& \sqsupseteq\sqsubseteq y \leftarrow \bullet; x \leftarrow \mho; \mathtt{ret}\ \mathtt{absurd}\ x && \text{(comm conv)} \\
& \sqsupseteq\sqsubseteq y \leftarrow \bullet; \mho && \text{(Strictness of Stacks)} \\
& \sqsubseteq y \leftarrow \bullet; \mathtt{ret}\ y && (\mho \text{ is } \bot) \\
& \sqsupseteq\sqsubseteq \bullet && (\underline{F}\eta)
\end{aligned}$$

3. $+$:

a) Retraction is

$$\begin{aligned}
& x : A_1 + A_2 \vdash \\
& [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle]\!][\mathtt{ret}\ [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][x]] \\
& = [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle]\!] \\
& \quad [\mathtt{ret}\ \mathtt{case}\ x \{ x_1.\mathtt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1] \mid x_1.\mathtt{inr}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2] \}] \\
& \sqsupseteq\sqsubseteq \mathtt{case}\ x && \text{(commuting conversion)} \\
& \quad \{ x_1.[\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle]\!][\mathtt{ret}\ \mathtt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]] \\
& \quad \mid x_2.[\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle]\!][\mathtt{ret}\ \mathtt{inr}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]] \} \\
& \sqsupseteq\sqsubseteq \mathtt{case}\ x && \text{(cast computation)} \\
& \quad \{ x_1.x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!]x_1]; \mathtt{ret}\ \mathtt{inl}\ x_1 \\
& \quad \mid x_2.x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!]x_2]; \mathtt{ret}\ \mathtt{inr}\ x_2 \} \\
& \sqsupseteq\sqsubseteq \mathtt{case}\ x \{ x_1.\mathtt{ret}\ \mathtt{inl}\ x_1 \mid x_2.\mathtt{ret}\ \mathtt{inr}\ x_2 \} \\
& && \text{(IH retraction)} \\
& \sqsupseteq\sqsubseteq \mathtt{ret}\ x && (+\eta)
\end{aligned}$$

b) For Projection:

$$\begin{aligned}
& \bullet : A_1' + A_2' \vdash \\
& x \leftarrow [\![\langle \underline{F}(A_1 + A_2) \swarrow \underline{F}(A_1' + A_2') \rangle]\!]; [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][x] \\
& = x \leftarrow (x' \leftarrow \bullet; \mathtt{case}\ x' \{ x_1'.x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; \mathtt{ret}\ \mathtt{inl}\ x_1 \mid x_2'.\cdots \}); \\
& \quad [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!] \\
& \sqsupseteq\sqsubseteq x \leftarrow \bullet;' \mathtt{case}\ x' && \text{(Commuting Conversion)} \\
& \quad \{ x_1'.x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!]\mathtt{ret}\ \mathtt{inl}\ x_1 \\
& \quad \mid x_2'.x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ x_2']; [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!]\mathtt{ret}\ \mathtt{inr}\ x_2 \} \\
& \sqsupseteq\sqsubseteq x \leftarrow \bullet;' \mathtt{case}\ x' && \text{(Cast Computation)} \\
& \quad \{ x_1'.x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; \mathtt{ret}\ \mathtt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!]x_1 \\
& \quad \mid x_2'.x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ x_2']; \mathtt{ret}\ \mathtt{inr}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!]x_2 \} \\
& \sqsubseteq x \leftarrow \bullet;' \mathtt{case}\ x' \{ x_1'.\mathtt{ret}\ \mathtt{inl}\ x_1' \mid x_2'.\mathtt{ret}\ \mathtt{inr}\ x_2' \} \\
& && \text{(IH projection)} \\
& \sqsupseteq\sqsubseteq x \leftarrow \bullet;' \mathtt{ret}\ x' && (+\eta) \\
& \sqsupseteq\sqsubseteq \bullet && (\underline{F}\eta)
\end{aligned}$$

4. $\times$:
   a) First, Retraction:

$$
\begin{aligned}
&x : A_1 \times A_2 \vdash \\
&[\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle]\!][\mathtt{ret}\ [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!][x]] \\
&= [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle]\!] \\
&\quad [\mathtt{ret}\ \mathtt{let}\ (x_1, x_2) = x; ([\![\langle A_1' \nwarrow A_1\rangle]\!][x_1], [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2])] \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x_1, x_2) = x; [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2')\rangle]\!] \\
&\qquad [\mathtt{ret}\ ([\![\langle A_1' \nwarrow A_1\rangle]\!][x_1], [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2])] \\
&\qquad\qquad \text{(commuting conversion)} \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x_1, x_2) = x; \qquad \text{(cast reduction)} \\
&\quad y_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1'\rangle]\!][\mathtt{ret}\ [\![\langle A_1' \nwarrow A_1\rangle]\!][x_1]]; \\
&\quad y_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2'\rangle]\!][\mathtt{ret}\ [\![\langle A_2' \nwarrow A_2\rangle]\!][x_2]]; \\
&\quad \mathtt{ret}\ (y_1, y_2) \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x_1, x_2) = x; y_1 \leftarrow \mathtt{ret}\ x_1; y_2 \leftarrow \mathtt{ret}\ x_2; \mathtt{ret}\ (y_1, y_2) \\
&\qquad\qquad \text{(IH retraction)} \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x_1, x_2) = x; \mathtt{ret}\ (x_1, x_2) \qquad (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq \mathtt{ret}\ x \qquad (\times\eta)
\end{aligned}
$$

b) Next, Projection:

$$
\begin{array}{ll}
\bullet : \underline{F}A' \vdash & \\
x \leftarrow [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2') \rangle]\!][\bullet]; \mathtt{ret}\ [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][x] & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & (\underline{F}\eta, \times\eta) \\
\quad x \leftarrow [\![\langle \underline{F}(A_1 \times A_2) \swarrow \underline{F}(A_1' \times A_2') \rangle]\!][\mathtt{ret}\ (x_1', x_2')]; & \\
\quad \mathtt{ret}\ [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][x] & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & \text{(cast reduction)} \\
\quad x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; & \\
\quad x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ x_2']; & \\
\quad \mathtt{ret}\ [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][(x_1, x_2)] & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & \text{(cast reduction)} \\
\quad x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; & \\
\quad x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ x_2']; & \\
\quad \mathtt{ret}\ ([\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1], [\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]) & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & (\underline{F}\beta, \text{twice}) \\
\quad x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; & \\
\quad x_2 \leftarrow [\![\langle \underline{F}A_2 \swarrow \underline{F}A_2' \rangle]\!][\mathtt{ret}\ x_2']; & \\
\quad y_2' \leftarrow \mathtt{ret}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]; & \\
\quad y_1' \leftarrow \mathtt{ret}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]; & \\
\quad \mathtt{ret}\ (y_1', y_2') & \\
\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & \text{(IH Projection)} \\
\quad x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; & \\
\quad y_2' \leftarrow \mathtt{ret}\ x_2'; & \\
\quad y_1' \leftarrow \mathtt{ret}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]; & \\
\quad \mathtt{ret}\ (y_1', y_2') & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & (\underline{F}\beta) \\
\quad x_1 \leftarrow [\![\langle \underline{F}A_1 \swarrow \underline{F}A_1' \rangle]\!][\mathtt{ret}\ x_1']; & \\
\quad y_1' \leftarrow \mathtt{ret}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]; & \\
\quad \mathtt{ret}\ (x_1', y_2') & \\
\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; & \text{(IH Projection)} \\
\quad y_1' \leftarrow \mathtt{ret}\ x_1'; & \\
\quad \mathtt{ret}\ (x_1', y_2') & \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{let}\ (x_1', x_2') = x'; \mathtt{ret}\ (x_1', x_2') & (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq\ x' \leftarrow \bullet; \mathtt{ret}\ x' & (\times\eta) \\
\sqsupseteq\sqsubseteq\ \bullet & (\underline{F}\eta)
\end{array}
$$

5. U: By inductive hypothesis, $(x.[\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!], \langle \underline{B} \swarrow \underline{B}' \rangle)$ is a computation ep pair

   a) To show retraction we need to prove:

$x : U\underline{B} \vdash \texttt{ret}\ x \sqsupseteq\sqsubseteq y \leftarrow (\texttt{ret thunk}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!]); \texttt{ret thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\texttt{force}\ y]$

Which we calculate as follows:

$$
\begin{aligned}
& x : U\underline{B} \vdash \\
& [\![\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle]\!][(\texttt{ret}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x])] \\
& \sqsupseteq\sqsubseteq \texttt{ret thunk}\ ([\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\texttt{force}\ [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x]]) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Cast Reduction)} \\
& \sqsupseteq\sqsubseteq \texttt{ret thunk force}\ x \qquad\qquad\qquad\qquad\qquad\qquad \text{(IH Retraction)} \\
& \sqsupseteq\sqsubseteq \texttt{ret}\ x \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\eta)
\end{aligned}
$$

b) To show projection we calculate:

$$
\begin{aligned}
& x \leftarrow [\![\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle]\!][\bullet]; [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x] \\
& \sqsupseteq\sqsubseteq x' \leftarrow \bullet; x \leftarrow [\![\langle \underline{F}U\underline{B} \swarrow \underline{F}U\underline{B}'\rangle]\!][\texttt{ret}\ x']; [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\eta) \\
& \sqsupseteq\sqsubseteq x' \leftarrow \bullet; x \leftarrow \texttt{ret thunk}\ ([\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\texttt{force}\ x']); [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Cast Reduction)} \\
& \sqsupseteq\sqsubseteq x' \leftarrow \bullet; [\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][\texttt{thunk}\ ([\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\texttt{force}\ x'])] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\beta) \\
& \sqsubseteq x' \leftarrow \bullet; x' \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(IH Projection)} \\
& \sqsupseteq\sqsubseteq \bullet \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\eta)
\end{aligned}
$$

1. There's a few base cases about the dynamic computation type, then

2. $\top$:

   a) Retraction is by $\top\eta$:

   $$z : U\top \vdash \texttt{force}\ z \sqsupseteq\sqsubseteq \{\} : \top$$

   b) Projection is

   $$
   \begin{aligned}
   & \texttt{thunk}\ \mho \sqsubseteq \texttt{thunk force}\ w \qquad\qquad\qquad\qquad (\mho \text{ is } \bot) \\
   & \qquad\qquad \sqsupseteq\sqsubseteq w \qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\eta)
   \end{aligned}
   $$

3. &: Retraction

$$
\begin{aligned}
&z : U(\underline{B}_1 \,\&\, \underline{B}_2) \vdash \\
&[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][z]] \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][z]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\&\eta) \\
&\qquad \mid \pi' \mapsto \pi'[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][z]]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto [\![\langle \underline{B}_1 \nleftarrow \underline{B}_1' \rangle]\!][\pi\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][z]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Cast reduction)} \\
&\qquad \mid \pi' \mapsto [\![\langle \underline{B}_2 \nleftarrow \underline{B}_2' \rangle]\!][\pi'\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][z]]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto [\![\langle \underline{B}_1 \nleftarrow \underline{B}_1' \rangle]\!][\mathtt{force}\ [\![\langle U\underline{B}_1' \leftharpoonup U\underline{B}_1 \rangle]\!][\mathtt{thunk}\ \pi\mathtt{force}\ z]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Cast reduction)} \\
&\qquad \mid \pi' \mapsto [\![\langle \underline{B}_2 \nleftarrow \underline{B}_2' \rangle]\!][\mathtt{force}\ [\![\langle U\underline{B}_2' \leftharpoonup U\underline{B}_2 \rangle]\!][\mathtt{thunk}\ \pi'\mathtt{force}\ z]]\} \\
&\sqsupseteq\sqsubseteq (\mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ z, \mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ z) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(IH retraction)} \\
&\sqsupseteq\sqsubseteq (\pi\mathtt{force}\ z, \pi'\mathtt{force}\ z) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\beta) \\
&\sqsupseteq\sqsubseteq \mathtt{force}\ z \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\&\eta)
\end{aligned}
$$

Projection

$$
\begin{aligned}
&w : U\underline{B}_1' \,\&\, \underline{B}_2' \vdash \\
&[\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]] \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\eta) \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \pi\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]] \\
&\qquad \mid \pi' \mapsto \pi'\mathtt{force}\ [\![\langle U(\underline{B}_1' \,\&\, \underline{B}_2') \leftharpoonup U(\underline{B}_1 \,\&\, \underline{B}_2)\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]]\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\&\eta) \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ [\![\langle U\underline{B}_1' \leftharpoonup U\underline{B}_1 \rangle]\!][\mathtt{thunk}\ \pi\mathtt{force}\ \mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]] \\
&\qquad \mid \pi' \mapsto \mathtt{force}\ [\![\langle U\underline{B}_2' \leftharpoonup U\underline{B}_2 \rangle]\!][\mathtt{thunk}\ \pi'\mathtt{force}\ \mathtt{thunk}\ [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]]\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(cast reduction)} \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ [\![\langle U\underline{B}_1' \leftharpoonup U\underline{B}_1 \rangle]\!][\mathtt{thunk}\ \pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\beta) \\
&\qquad \mid \pi' \mapsto \mathtt{force}\ [\![\langle U\underline{B}_2' \leftharpoonup U\underline{B}_2 \rangle]\!][\mathtt{thunk}\ \pi'[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \nleftarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][\mathtt{force}\ w]]\} \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ [\![\langle U\underline{B}_1' \leftharpoonup U\underline{B}_1 \rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \nleftarrow \underline{B}_1' \rangle]\!][\pi\mathtt{force}\ w]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(cast reduction)} \\
&\qquad \mid \pi' \mapsto \mathtt{force}\ [\![\langle U\underline{B}_2' \leftharpoonup U\underline{B}_2 \rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_2 \nleftarrow \underline{B}_2' \rangle]\!][\pi'\mathtt{force}\ w]]\} \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \{\pi \mapsto \mathtt{force}\ [\![\langle U\underline{B}_1' \leftharpoonup U\underline{B}_1 \rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_1 \nleftarrow \underline{B}_1' \rangle]\!][\mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ w]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\beta) \\
&\qquad \mid \pi' \mapsto \mathtt{force}\ [\![\langle U\underline{B}_2' \leftharpoonup U\underline{B}_2 \rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B}_2 \nleftarrow \underline{B}_2' \rangle]\!][\mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ w]]\} \\
&\sqsubseteq \mathtt{thunk}\ (\mathtt{force}\ \mathtt{thunk}\ \pi\mathtt{force}\ w, \mathtt{force}\ \mathtt{thunk}\ \pi'\mathtt{force}\ w) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(IH projection)} \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ (\pi\mathtt{force}\ w, \pi'\mathtt{force}\ w) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\beta) \\
&\sqsupseteq\sqsubseteq \mathtt{thunk}\ \mathtt{force}\ w \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\&\eta) \\
&\sqsupseteq\sqsubseteq w \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\eta)
\end{aligned}
$$

4. $\to$: Retraction

$$
\begin{array}{ll}
z : U(A \to \underline{B}) \vdash & \\
[\![\langle A \to \underline{B} \nleftarrow A' \to \underline{B}' \rangle]\!][\mathtt{force}\ [\![\langle U(A' \to \underline{B}') \leftarrow U(A \to \underline{B})\rangle]\!][z]] & \\
\sqsupseteq\sqsubseteq\ \lambda x : A.([\![\langle A \to \underline{B} \nleftarrow A' \to \underline{B}' \rangle]\!][\mathtt{force}\ [\![\langle U(A' \to \underline{B}') \leftarrow U(A \to \underline{B})\rangle]\!][z]])\ x & (\to \eta) \\
\sqsupseteq\sqsubseteq\ \lambda x : A.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][(\mathtt{force}\ [\![\langle U(A' \to \underline{B}') \leftarrow U(A \to \underline{B})\rangle]\!][z])([\![\langle A' \leftarrow A\rangle]\!][x])] & \text{(cast reduction)} \\
\sqsupseteq\sqsubseteq\ \lambda x : A. & \text{(cast reduction)} \\
\quad [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][y \leftarrow [\![\langle \underline{F}A \nleftarrow \underline{F}A' \rangle]\!][\mathtt{ret}\ \langle A' \leftarrow A\rangle[x]]; \mathtt{force}\ \langle U\underline{B}' \leftarrow U\underline{B}\rangle[\mathtt{thunk}\ ((\mathtt{force}\ z)\ y)]] & \\
\sqsupseteq\sqsubseteq\ \lambda x : A.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][y \leftarrow \mathtt{ret}\ x; \mathtt{force}\ \langle U\underline{B}' \leftarrow U\underline{B}\rangle[\mathtt{thunk}\ ((\mathtt{force}\ z)\ y)]] & \text{(IH Retraction)} \\
\sqsupseteq\sqsubseteq\ \lambda x : A.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{force}\ \langle U\underline{B}' \leftarrow U\underline{B}\rangle[\mathtt{thunk}\ ((\mathtt{force}\ z)\ x)]] & (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq\ \lambda x : A.\mathtt{force}\ \mathtt{thunk}\ ((\mathtt{force}\ z)\ x) & \text{(IH retraction)} \\
\sqsupseteq\sqsubseteq\ \lambda x : A.(\mathtt{force}\ z)\ x & (U\beta) \\
\sqsupseteq\sqsubseteq\ \mathtt{force}\ z & (\to \eta)
\end{array}
$$

Projection

$$
\begin{array}{ll}
& w : U(A' \to \underline{B}') \vdash \\
& [\![\langle U(A' \to \underline{B}') \leftharpoondown U(A \to \underline{B})\rangle]\!][\mathtt{thunk}\ [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!][\mathtt{force}\ w]] \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \mathtt{force}\ [\![\langle U(A' \to \underline{B}') \leftharpoondown U(A \to \underline{B})\rangle]\!][\mathtt{thunk}\ [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!][\mathtt{force}\ w]] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (U\eta) \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \\
& \quad (\mathtt{force}\ [\![\langle U(A' \to \underline{B}') \leftharpoondown U(A \to \underline{B})\rangle]\!][\mathtt{thunk}\ [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!][\mathtt{force}\ w]])\ x' \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\to \eta) \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \\
& \quad x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{ret}\ x']; \qquad\qquad \text{(cast reduction)} \\
& \quad \mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ ((\mathtt{force}\ \mathtt{thunk}\ [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!][\mathtt{force}\ w])\ x)] \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \\
& \quad x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{ret}\ x']; \qquad\qquad (U\beta) \\
& \quad \mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ (([\![\langle A \to \underline{B} \swarrow A' \to \underline{B}'\rangle]\!][\mathtt{force}\ w])\ x)] \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \\
& \quad x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{ret}\ x']; \qquad\qquad \text{(cast reduction)} \\
& \quad \mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\ (\langle A' \leftharpoondown A\rangle[x])]] \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \\
& \quad x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{ret}\ x']; \qquad\qquad (\underline{F}\beta) \\
& \quad x' \leftarrow \mathtt{ret}\ \langle A' \leftharpoondown A\rangle[x]; \\
& \quad \mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\ x']] \\
\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'. \qquad\qquad \text{(IH projection)} \\
& \quad x' \leftarrow \mathtt{ret}\ x'; \\
& \quad \mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\ x']] \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][(\mathtt{force}\ w)\ x']] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ [\![\langle U\underline{B}' \leftharpoondown U\underline{B}\rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\mathtt{force}\ \mathtt{thunk}\ ((\mathtt{force}\ w)\ x')]] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{F}\beta) \\
\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'.\mathtt{force}\ \mathtt{thunk}\ ((\mathtt{force}\ w)\ x') \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(IH projection)} \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \lambda x' : A'.((\mathtt{force}\ w)\ x') \qquad\qquad (U\beta) \\
\sqsupseteq\sqsubseteq & \mathtt{thunk}\ \mathtt{force}\ w \qquad\qquad (\to \eta) \\
\sqsupseteq\sqsubseteq & w \qquad\qquad (U\eta)
\end{array}
$$

5. $\underline{F}$:

   a) To show retraction we need to show

$$z : U\underline{F}A \vdash \mathtt{force}\ z \sqsupseteq\sqsubseteq [\![\langle \underline{F}A \swarrow \underline{F}A'\rangle]\!][\mathtt{force}\ \mathtt{thunk}\ (x \leftarrow \mathtt{force}\ z; \mathtt{ret}\ [\![\langle A' \leftharpoondown A\rangle]\!])]$$

We calculate:

$$\begin{array}{lr}
[\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][\texttt{force thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!])] & \\
\sqsupseteq\sqsubseteq [\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][(x \leftarrow \texttt{force}\ z; \texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!])] & \\
& (U\beta) \\
\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; [\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][\texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!]] & \\
& \text{(comm conv)} \\
\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; \texttt{ret}\ x & \text{(IH value retraction)} \\
\sqsupseteq\sqsubseteq \texttt{force}\ z & (\underline{F}\eta)
\end{array}$$

b) To show projection we need to show

$$w : U\underline{F}A' \vdash \texttt{thunk}\ (x \leftarrow \texttt{force thunk}\ [\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][\texttt{force}\ w]; \texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!]) \sqsubseteq w$$

We calculate as follows

$$\begin{array}{lr}
\texttt{thunk}\ (x \leftarrow \texttt{force thunk}\ [\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][\texttt{force}\ w]; \texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!]) & \\
\sqsupseteq\sqsubseteq \texttt{thunk}\ (x \leftarrow [\![\langle \underline{F}A \not\leftarrow \underline{F}A' \rangle]\!][\texttt{force}\ w]; \texttt{ret}\ [\![\langle A' \not\rightarrow A \rangle]\!]) & \\
& (U\beta) \\
\sqsubseteq \texttt{thunk force}\ w & \text{(IH value projection)} \\
\sqsupseteq\sqsubseteq w & (U\eta)
\end{array}$$

□

While tedious, this work pays off greatly in later proofs: this is the *only* proof in the entire development that needs to inspect the definition of a "shifted" cast (a downcast between $\underline{F}$ types or an upcast between U types). All later lemmas have cases for these shifted casts, but *only* use the property that they are part of an ep pair. This is one of the biggest advantages of using an explicit syntax for complex values and complex stacks: the shifted casts are the only ones that non-trivially use effectful terms, so after this lemma is established we only have to manipulate values and stacks, which compose much more nicely than effectful terms. Conceptually, the main reason we can avoid reasoning about the definitions of the shifted casts directly is that any two shifted casts that form an ep pair with the same value embedding/stack projection are equal:

Lemma 133 (Embedding determines Projection, and vice-versa)**.** *For any value $x : A \vdash V_e : A'$ and stacks $\bullet : \underline{F}A' \vdash S_1 : \underline{F}A$ and $\bullet : \underline{F}A' \vdash S_2 : \underline{F}A$, if (V_e, S_1) and (V_e, S_2) are both value ep pairs, then*

$$S_1 \sqsupseteq\sqsubseteq S_2$$

Similarly for any values $x : U\underline{B} \vdash V_1 : U\underline{B}'$ and $x : U\underline{B} \vdash V_2 : U\underline{B}'$ and stack $\bullet : \underline{B}' \vdash S_p : \underline{B}$, if (V_1, S_p) and (V_2, S_p) are both computation ep pairs then

$$V_1 \sqsupseteq\sqsubseteq V_2$$

Proof. By symmetry it is sufficient to show $S_1 \sqsubseteq S_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{S_1 \sqsubseteq S_1}{x \leftarrow S_1; \mathtt{ret}\ x \sqsubseteq x \leftarrow \bullet; S_1[\mathtt{ret}\ x]}}{x \leftarrow S_1; \mathtt{ret}\ V_e \sqsubseteq x \leftarrow \bullet; \mathtt{ret}\ x}}{x \leftarrow S_1; \mathtt{ret}\ x \sqsubseteq x \leftarrow \bullet; S_2[\mathtt{ret}\ x]}}{\bullet : \underline{F}A' \vdash S_1 \sqsubseteq S_2 : \underline{F}A}$$

similarly to show $V_1 \sqsubseteq V_2$:

$$\dfrac{\dfrac{\dfrac{x : U\underline{B} \vdash \mathtt{thunk}\ \mathtt{force}\ V_2 \sqsubseteq \mathtt{thunk}\ \mathtt{force}\ V_2 : U\underline{B}'}{x : U\underline{B} \vdash \mathtt{thunk}\ \mathtt{force}\ x \sqsubseteq \mathtt{thunk}\ S_p[\mathtt{force}\ V_2]}}{x : U\underline{B} \vdash \mathtt{thunk}\ \mathtt{force}\ V_1 \sqsubseteq \mathtt{thunk}\ \mathtt{force}\ V_2 : U\underline{B}'}}{x : U\underline{B} \vdash V_1 \sqsubseteq V_2 : U\underline{B}'}$$

□

The next two lemmas on the way to axiomatic graduality show that Figure 7.5 translates $\langle A \nwarrow A \rangle$ to the identity and $\langle A'' \nwarrow A' \rangle \langle A' \nwarrow A \rangle$ to the same contract as $\langle A'' \nwarrow A \rangle$, and similarly for downcasts. Intuitively, for all connectives except $\underline{F}, U$, this is because of functoriality of the type constructors on values and stacks. For the $\underline{F}, U$ cases, we will use the corresponding fact about the dual cast, i.e., to prove the $\underline{F}A$ to $\underline{F}A$ downcast is the identity stack, we know by inductive hypothesis that the A to A upcast is the identity, and that the identity stack is a projection for the identity. Therefore Lemma 133 implies that the $\underline{F}A$ downcast must be equivalent to the identity. We now discuss these two lemmas and their proofs in detail.

First, we show that the casts from a type to itself are equivalent to the identity. Below, we will use this lemma to prove the reflexivity case of the axiomatic graduality theorem, and to prove a conservativity result, which says that a GTT homogeneous term precision is the same as a CBPV* inequality between their translations.

Lemma 134 (Identity Expansion)**.** *For any A and $\underline{B}$,*

$$x : A \vdash [\![\langle A \nwarrow A \rangle]\!] \sqsupseteq\sqsubseteq x : A \qquad \bullet : \underline{B} \vdash [\![\langle \underline{B} \swarrow \underline{B} \rangle]\!] \sqsupseteq\sqsubseteq \bullet : \underline{B}$$

Proof. We proceed by induction on $A, \underline{B}$, following the proof that reflexivity is admissible given in Lemma 125.

1. If $A \in \{1, ?\}$, then $[\![\langle A \nwarrow A \rangle]\!][x] = x$.
2. If $A = 0$, then $\mathtt{absurd}\ x \sqsupseteq\sqsubseteq x$ by 0η.
3. If $A = U\underline{B}$, then by inductive hypothesis $[\![\langle \underline{B} \swarrow \underline{B} \rangle]\!] \sqsupseteq\sqsubseteq \bullet$. By Lemma 130, $(x.x, \bullet)$ is a computation ep pair from $\underline{B}$ to itself. But by Lemma 132, $([\![\langle U\underline{B} \nwarrow U\underline{B} \rangle]\!][x], \bullet)$ is also a computation ep pair so the result follows by uniqueness of embeddings from computation projections Lemma 133.

4. If $A = A_1 \times A_2$ or $A = A_1 + A_2$, the result follows by the η principle and inductive hypothesis.
5. If $\underline{B} = \underline{¿}$, $[\![\langle \underline{¿} \nleftarrow \underline{¿} \rangle]\!] = \bullet$.
6. For $\underline{B} = \top$, the result follows by $\top\eta$.
7. For $\underline{B} = \underline{B}_1 \,\&\, \underline{B}_2$ or $\underline{B} = A \to \underline{B}'$, the result follows by inductive hypothesis and η.
8. For $\underline{B} = \underline{F}A$, by inductive hypothesis, the downcast is a projection for the value embedding $x.x$, so the result follows by identity ep pair and uniqueness of projections from value embeddings.

□

Second, we show that a composition of upcasts is translated to the same thing as a direct upcast, and similarly for downcasts. Below, we will use this lemma to translate *transitivity* of term precision in GTT.

Lemma 135 (Cast Decomposition)**.** *For any dynamic type interpretation ρ,*

$$\frac{A \sqsubseteq A' \sqsubseteq A''}{x : A \vdash [\![\langle A'' \leftarrow A \rangle]\!]_\rho \sqsupseteq\sqsubseteq [\![\langle A'' \leftarrow A' \rangle]\!]_\rho[[\![\langle A' \leftarrow A \rangle]\!]_\rho] : A''}$$

$$\frac{\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''}{\bullet : \underline{B}'' \vdash [\![\langle \underline{B} \nleftarrow \underline{B}'' \rangle]\!]_\rho \sqsupseteq\sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!]_\rho[[\![\langle \underline{B}' \nleftarrow \underline{B}'' \rangle]\!]_\rho]}$$

Proof. By mutual induction on $A, \underline{B}$.

1. $A \sqsubseteq A' \sqsubseteq A''$
   a) If $A = 0$, we need to show $x : 0 \vdash [\![\langle A'' \leftarrow 0 \rangle]\!][x] \sqsupseteq\sqsubseteq [\![\langle A'' \leftarrow A' \rangle]\!][[\![\langle A' \leftarrow 0 \rangle]\!][x]] : A''$ which follows by 0η.
   b) If $A = ?$, then $A' = A'' = ?$, and both casts are the identity.
   c) If $A \notin \{?, 0\}$ and $A' = ?$, then $A'' = ?$ and $[\![\langle ? \leftarrow ? \rangle]\!][[\![\langle ? \leftarrow A \rangle]\!]] = [\![\langle ? \leftarrow A \rangle]\!]$ by definition.
   d) If $A, A' \notin \{?, 0\}$ and $A'' = ?$, then $\lfloor A \rfloor = \lfloor A' \rfloor$, which we call G and

$$[\![\langle ? \leftarrow A \rangle]\!] = [\![\langle ? \leftarrow G \rangle]\!][[\![\langle G \leftarrow A \rangle]\!]]$$

   and

$$[\![\langle ? \leftarrow A' \rangle]\!][[\![\langle A' \leftarrow A \rangle]\!]] = [\![\langle ? \leftarrow G \rangle]\!][[\![\langle G \leftarrow A' \rangle]\!][[\![\langle A' \leftarrow A \rangle]\!]]]$$

   so this reduces to the case for $A \sqsubseteq A' \sqsubseteq G$, below.
   e) If $A, A', A'' \notin \{?, 0\}$, then they all have the same top-level constructor:

i. $+$: We need to show for $A_1 \sqsubseteq A_1' \sqsubseteq A_1''$ and $A_2 \sqsubseteq A_2' \sqsubseteq A_2''$:

$$x : [\![A_1]\!] + [\![A_2]\!] \vdash [\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][x]] \\ \sqsupseteq\sqsubseteq [\![\langle A_1'' + A_2'' \nwarrow A_1 + A_2 \rangle]\!][x] : [\![A_1'']\!] + [\![A_2'']\!]$$

We proceed as follows:

$$\begin{array}{ll} & [\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][x]] \\ \sqsupseteq\sqsubseteq & \texttt{case}\ x & (+\eta) \\ & \quad \{x_1.[\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][\texttt{inl}\ x_1]] \\ & \quad \mid x_2.[\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][\texttt{inr}\ x_2]]\} \\ \sqsupseteq\sqsubseteq & \texttt{case}\ x & (\text{cast reduction}) \\ & \quad \{x_1.[\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][\texttt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]] \\ & \quad \mid x_2.[\![\langle A_1'' + A_2'' \nwarrow A_1' + A_2' \rangle]\!][\texttt{inr}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]]\} \\ \sqsupseteq\sqsubseteq & \texttt{case}\ x & (\text{cast reduction}) \\ & \quad \{x_1.\texttt{inl}\ [\![\langle A_1'' \nwarrow A_1' \rangle]\!][[\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]] \\ & \quad \mid x_2.\texttt{inr}\ [\![\langle A_2'' \nwarrow A_2' \rangle]\!][[\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]]\} \\ \sqsupseteq\sqsubseteq & \texttt{case}\ x & (\text{IH}) \\ & \quad \{x_1.\texttt{inl}\ [\![\langle A_1'' \nwarrow A_1 \rangle]\!][x_1] \\ & \quad \mid x_2.\texttt{inr}\ [\![\langle A_2'' \nwarrow A_2 \rangle]\!][x_2]\} \\ = & [\![\langle A_1'' + A_2'' \nwarrow A_1 + A_2 \rangle]\!][x] & (\text{definition}) \end{array}$$

ii. 1: By definition both sides are the identity.

iii. $\times$: We need to show for $A_1 \sqsubseteq A_1' \sqsubseteq A_1''$ and $A_2 \sqsubseteq A_2' \sqsubseteq A_2''$:

$$x : [\![A_1]\!] \times [\![A_2]\!] \vdash [\![\langle A_1'' \times A_2'' \nwarrow A_1' \times A_2' \rangle]\!][[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][x]] \\ \sqsupseteq\sqsubseteq [\![\langle A_1'' \times A_2'' \nwarrow A_1 \times A_2 \rangle]\!][x] : [\![A_1'']\!] \times [\![A_2'']\!].$$

We proceed as follows:

$$\begin{array}{ll} & [\![\langle A_1'' \times A_2'' \nwarrow A_1' \times A_2' \rangle]\!][[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][x]] \\ \sqsupseteq\sqsubseteq & \texttt{let}\ (y,z) = x; [\![\langle A_1'' \times A_2'' \nwarrow A_1' \times A_2' \rangle]\!][[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2 \rangle]\!][(y,z)]] \\ & & (\times\eta) \\ \sqsupseteq\sqsubseteq & \texttt{let}\ (y,z) = x; [\![\langle A_1'' \times A_2'' \nwarrow A_1' \times A_2' \rangle]\!][([\![\langle A_1' \nwarrow A_1 \rangle]\!][y], [\![\langle A_2' \nwarrow A_2 \rangle]\!][z])] \\ & & (\text{cast reduction}) \\ \sqsupseteq\sqsubseteq & \texttt{let}\ (y,z) = x; ([\![\langle A_1'' \nwarrow A_1' \rangle]\!][[\![\langle A_1' \nwarrow A_1 \rangle]\!][y]], [\![\langle A_2'' \nwarrow A_2' \rangle]\!][[\![\langle A_2' \nwarrow A_2 \rangle]\!][z]]) \\ & & (\text{cast reduction}) \\ \sqsupseteq\sqsubseteq & \texttt{let}\ (y,z) = x; ([\![\langle A_1'' \nwarrow A_1 \rangle]\!][y], [\![\langle A_2'' \nwarrow A_2 \rangle]\!][z]) \\ & & (\text{IH}) \\ = & [\![\langle A_1'' \times A_2'' \nwarrow A_1 \times A_2 \rangle]\!][x] & (\text{definition}) \end{array}$$

iv. $U\underline{B} \sqsubseteq U\underline{B}' \sqsubseteq U\underline{B}''$. We need to show

$$x : U\underline{B} \vdash [\![\langle U\underline{B}'' \nwarrow U\underline{B}' \rangle]\!][[\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!][x]] \sqsupseteq\sqsubseteq [\![\langle U\underline{B}'' \nwarrow U\underline{B} \rangle]\!][x] : U\underline{B}''$$

By composition of ep pairs, we know

$$(x.[\![\langle U\underline{B}'' \nwarrow U\underline{B}'\rangle]\!][[\![\langle U\underline{B}' \nwarrow U\underline{B}\rangle]\!][x]], [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![\langle \underline{B}' \swarrow \underline{B}''\rangle]\!]])$$

is a computation ep pair. Furthermore, by inductive hypothesis, we know

$$[\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![\langle \underline{B}' \swarrow \underline{B}''\rangle]\!]] \sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}''\rangle]\!]$$

so then both sides form ep pairs paired with $[\![\langle \underline{B} \swarrow \underline{B}''\rangle]\!]$, so it follows because computation projections determine embeddings Lemma 133.

2. $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$
   a) If $\underline{B} = \top$, then the result is immediate by $\eta\top$.
   b) If $\underline{B} = \underline{\iota}$, then $\underline{B}' = \underline{B}'' = \underline{\iota}$ then both sides are just $\bullet$.
   c) If $\underline{B} \notin \{\underline{\iota}, \top\}$, and $\underline{B}' = \underline{\iota}$, then $\underline{B}'' = \underline{\iota}$

$$[\![\langle \underline{B} \swarrow \underline{\iota}\rangle]\!][[\![\langle \underline{\iota} \swarrow \underline{\iota}\rangle]\!]] = [\![\langle \underline{B} \swarrow \underline{\iota}\rangle]\!]$$

   d) If $\underline{B}, \underline{B}' \notin \{\underline{\iota}, \top\}$, and $\underline{B}'' = \underline{\iota}$, and $\lfloor \underline{B} \rfloor = \lfloor \underline{B}' \rfloor$, which we call $\underline{G}$. Then we need to show

$$[\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![\langle \underline{B}' \swarrow \underline{G}\rangle]\!][[\![\langle \underline{G} \swarrow \underline{\iota}\rangle]\!]]] \sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{G}\rangle]\!][[\![\langle \underline{G} \swarrow [\rangle]\!]\underline{\iota}]]$$

   so the result follows from the case $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{G}$, which is handled below.

   e) If $\underline{B}, \underline{B}', \underline{B}'' \notin \{\underline{\iota}, \top\}$, then they all have the same top-level constructor:
      i. & We are given $\underline{B}_1 \sqsubseteq \underline{B}_1' \sqsubseteq \underline{B}_1''$ and $\underline{B}_2 \sqsubseteq \underline{B}_2' \sqsubseteq \underline{B}_2''$ and we need to show

$$\bullet : \underline{B}_1'' \,\&\, \underline{B}_2'' \vdash [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!][[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]] : \underline{B}_1 \,\&\, \underline{B}_2$$

We proceed as follows:

$$\begin{aligned}
&[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!][[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]] \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!][[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\&\eta) \\
&\quad \mid \pi' \mapsto \pi'[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2'\rangle]\!][[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto [\![\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle]\!][\pi[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(cast reduction)} \\
&\quad \mid \pi' \mapsto [\![\langle \underline{B}_2 \swarrow \underline{B}_2'\rangle]\!][\pi'[\![\langle \underline{B}_1' \,\&\, \underline{B}_2' \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!]]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto [\![\langle \underline{B}_1 \swarrow \underline{B}_1'\rangle]\!][[\![\langle \underline{B}_1' \swarrow \underline{B}_1''\rangle]\!][\pi\bullet]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(cast reduction)} \\
&\quad \mid \pi' \mapsto [\![\langle \underline{B}_2 \swarrow \underline{B}_2'\rangle]\!][\langle \underline{B}_2' \swarrow \underline{B}_2''\rangle]\!][\pi'\bullet]\} \\
&\sqsupseteq\sqsubseteq ([\![\langle \underline{B}_1 \swarrow \underline{B}_1''\rangle]\!][\pi\bullet], [\![\langle \underline{B}_2 \swarrow \underline{B}_2''\rangle]\!][\pi'\bullet]) \quad \text{(IH)} \\
&= [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1'' \,\&\, \underline{B}_2''\rangle]\!] \qquad\qquad \text{(definition)}
\end{aligned}$$

ii. $\to$, assume we are given $A \sqsubseteq A' \sqsubseteq A''$ and $\underline{B} \sqsubseteq \underline{B}' \sqsubseteq \underline{B}''$, then we proceed:

$$\begin{array}{ll}
& [\![\langle A \to \underline{B} \leftarrow A' \to \underline{B}' \rangle]\!][[\![\langle A' \to \underline{B}' \leftarrow A'' \to \underline{B}'' \rangle]\!]] \\
\sqsupseteq\sqsubseteq & \lambda x : A.([\![\langle A \to \underline{B} \leftarrow A' \to \underline{B}' \rangle]\!][[\![\langle A' \to \underline{B}' \leftarrow A'' \to \underline{B}'' \rangle]\!]][\bullet])\, x \\
& \qquad\qquad (\to \eta) \\
\sqsupseteq\sqsubseteq & \lambda x : A.[\![\langle \underline{B} \leftarrow \underline{B}' \rangle]\!][([\![\langle A' \to \underline{B}' \leftarrow A'' \to \underline{B}'' \rangle]\!][\bullet])\, [\![\langle A' \nleftarrow A \rangle]\!][x]] \\
& \qquad\qquad \text{(cast reduction)} \\
\sqsupseteq\sqsubseteq & \lambda x : A.[\![\langle \underline{B} \leftarrow \underline{B}' \rangle]\!][[\![\langle \underline{B}' \leftarrow \underline{B}'' \rangle]\!][\bullet\, [\![\langle A'' \nleftarrow A' \rangle]\!][[\![\langle A' \nleftarrow A \rangle]\!][x]]]] \\
& \qquad\qquad \text{(cast reduction)} \\
\sqsupseteq\sqsubseteq & \lambda x : A.[\![\langle \underline{B} \leftarrow \underline{B}'' \rangle]\!][\bullet\, [\![\langle A'' \nleftarrow A \rangle]\!][x]] \\
= & [\![\langle A \to \underline{B} \leftarrow A \to \underline{B}'' \rangle]\!][\bullet] \qquad\qquad \text{(definition)}
\end{array}$$

iii. $\underline{F}A \sqsubseteq \underline{F}A' \sqsubseteq \underline{F}A''$. First, by composition of ep pairs, we know

$$(x.[\![\langle A'' \nleftarrow A' \rangle]\!][[\![\langle A' \nleftarrow A \rangle]\!][x]], [\![\langle \underline{F}A \leftarrow \underline{F}A' \rangle]\!])[[\![\langle \underline{F}A' \leftarrow \underline{F}A'' \rangle]\!]]$$

form a value ep pair. Furthermore, by inductive hypothesis, we know

$$x : A \vdash [\![\langle A'' \nleftarrow A' \rangle]\!][[\![\langle A' \nleftarrow A \rangle]\!][x]] \sqsupseteq\sqsubseteq [\![\langle A'' \nleftarrow A \rangle]\!][x]$$

so the two sides of our equation are both projections with the same value embedding, so the equation follows from uniqueness of projections from value embeddings.

□

The final lemma before the graduality theorem lets us "move a cast" from left to right or vice-versa, via the adjunction property for ep pairs. These arise in the proof cases for `return` and `thunk`, because in those cases the inductive hypothesis is in terms of an upcast (downcast) and the conclusion is in terms of a a downcast (upcast).

Lemma 136 (Hom-set formulation of Adjunction)**.** *For any value embedding-projection pair V_e, S_p from A to A', the following are equivalent:*

$$\frac{\Gamma \vdash \mathtt{ret}\ V_e[V] \sqsubseteq M : \underline{F}A'}{\Gamma \vdash \mathtt{ret}\ V \sqsubseteq S_p[M] : \underline{F}A}$$

For any computation ep pair (V_e, S_p) from $\underline{B}$ to $\underline{B}'$, the following are equivalent:

$$\frac{\Gamma, z' : U\underline{B}' \vdash M \sqsubseteq S[S_p[\mathtt{force}\ z']] : \underline{C}}{\Gamma, z : U\underline{B} \vdash M[V_e/z'] \sqsubseteq S[\mathtt{force}\ z] : \underline{C}}$$

Proof. 1. Assume $\mathtt{ret}\ V_e[V] \sqsubseteq M : \underline{F}A'$. Then by retraction, $\mathtt{ret}\ V \sqsubseteq S_p[\mathtt{ret}\ V_e[V]]$ so by transitivity, the result follows by substitution:

$$\frac{S_p \sqsubseteq S_p \qquad \mathtt{ret}\ V_e[V] \sqsubseteq M}{S_p[\mathtt{ret}\ V_e[V]] \sqsubseteq M}$$

2. Assume $\mathtt{ret}\ V \sqsubseteq S_p[M] : \underline{F}A$. Then by projection, $x \leftarrow S_p[M]; \mathtt{ret}\ V_e[x] \sqsubseteq M$, so it is sufficient to show

   $$\mathtt{ret}\ V_e[V] \sqsubseteq x \leftarrow S_p[M]; \mathtt{ret}\ V_e[x]$$

   but again by substitution we have

   $$x \leftarrow \mathtt{ret}\ V; \mathtt{ret}\ V_e[x] \sqsubseteq x \leftarrow S_p[M]; \mathtt{ret}\ V_e[x]$$

   and by $\underline{F}\beta$, the LHS is equivalent to $\mathtt{ret}\ V_e[V]$.

3. Assume $z' : U\underline{B'} \vdash M \sqsubseteq S[S_p[\mathtt{force}\ z']]$, then by projection, $S[S_p[\mathtt{force}\ V_e]] \sqsubseteq S[\mathtt{force}\ z]$ and by substitution:

   $$\frac{M \sqsubseteq S[S_p[\mathtt{force}\ z']] \qquad V_e \sqsubseteq V_e \qquad S[S_p[\mathtt{force}\ V_e]] = (S[S_p[\mathtt{force}\ z']])[V_e/z']}{M[V_e/z'] \sqsubseteq S[S_p[\mathtt{force}\ V_e]]}$$

4. Assume $z : U\underline{B} \vdash M[V_e/z'] \sqsubseteq S[\mathtt{force}\ z]$. Then by retraction, $M \sqsubseteq M[V_e[\mathtt{thunk}\ S_p[\mathtt{force}\ z]]]$ and by substitution:

   $$M[V_e[\mathtt{thunk}\ S_p[\mathtt{force}\ z]]] \sqsubseteq S[\mathtt{force}\ \mathtt{thunk}\ S_p[\mathtt{force}\ z]]$$

   and the right is equivalent to $S[S_p[\mathtt{force}\ z]]$ by $U\beta$.

□

Finally, we prove the axiomatic graduality theorem. In addition to the lemmas above, the main task is to prove the "compatibility" cases which are the congruence cases for introduction and elimination rules. These come down to proving that the casts "commute" with introduction/elimination forms, and are all simple calculations.

Theorem (Axiomatic Graduality)**.** For any dynamic type interpretation, the following are true:

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Psi : \Delta \sqsubseteq \Delta' \qquad \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B'}}{[\![\Gamma]\!] \mid [\![\Delta']\!] \vdash [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B'} \rangle]\!][[\![M']\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]}$$

$$\frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{[\![\Gamma]\!] \vdash [\![\langle A' \searrow A \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] : [\![A']\!]}$$

Proof. By mutual induction over term precision derivations. For the β, η and reflexivity rules, we use the identity expansion lemma and the corresponding β, η rule of CBPV*Lemma 134.

For compatibility rules a pattern emerges. Universal rules (positive intro, negative elim) are easy, we don't need to reason about casts at all. For "(co)-pattern matching rules" (positive elim, negative intro), we need to invoke the η principle (or commuting conversion, which is derived from the η principle). In all compatibility cases, the cast reduction lemma keeps the proof straightforward.

Fortunately, all reasoning about "shifted" casts is handled in lemmas, and here we only deal with the "nice" value upcasts/stack downcasts.

1. Transitivity for values: The GTT rule is

$$\frac{\begin{array}{c}\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi' : \Gamma' \sqsubseteq \Gamma'' \qquad \Phi'' : \Gamma \sqsubseteq \Gamma'' \\ \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A' \qquad \Phi' \vdash V' \sqsubseteq V'' : A' \sqsubseteq A''\end{array}}{\Phi'' \vdash V \sqsubseteq V'' : A \sqsubseteq A''}$$

   Which under translation (and the same assumptions about the contexts) is

$$\frac{\begin{array}{c}[\![\Gamma]\!] \vdash [\![\langle A' \nwarrow A\rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] : [\![A']\!] \\ [\![\Gamma']\!] \vdash [\![\langle A' \nwarrow A'\rangle]\!][[\![V']\!]] \sqsubseteq [\![V'']\!][[\![\Phi']\!]] : [\![A'']\!]\end{array}}{[\![\Gamma]\!] \vdash [\![\langle A'' \nwarrow A\rangle]\!][[\![V]\!]] \sqsubseteq [\![V'']\!][[\![\Phi'']\!]] : [\![A'']\!]}$$

   We proceed as follows, the key lemma here is the cast decomposition lemma:

$$\begin{aligned}[\![\langle A'' \nwarrow A\rangle]\!][[\![V]\!]] &\sqsupseteq\sqsubseteq [\![\langle A'' \nwarrow A'\rangle]\!][[\![\langle A' \nwarrow A\rangle]\!][[\![V]\!]]] && \text{(cast decomposition)}\\ &\sqsubseteq [\![\langle A'' \nwarrow A'\rangle]\!][[\![V']\!][[\![\Phi]\!]]] && \text{(IH)}\\ &\sqsubseteq [\![V'']\!][[\![\Phi']\!]][[\![\Phi]\!]] && \text{(IH)}\\ &\sqsupseteq\sqsubseteq [\![V'']\!][[\![\Phi'']\!]] && \text{(cast decomposition)}\end{aligned}$$

2. Transitivity for terms: The GTT rule is

$$\frac{\begin{array}{c}\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Phi' : \Gamma' \sqsubseteq \Gamma'' \qquad \Phi'' : \Gamma \sqsubseteq \Gamma'' \\ \Psi : \Delta \sqsubseteq \Delta' \qquad \Psi : \Delta' \sqsubseteq \Delta'' \qquad \Psi'' : \Delta \sqsubseteq \Delta'' \\ \Phi \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'\Phi' \mid \Psi' \vdash M' \sqsubseteq M'' : \underline{B}' \sqsubseteq \underline{B}''\end{array}}{\Phi'' \mid \Psi'' \vdash M \sqsubseteq M'' : \underline{B} \sqsubseteq \underline{B}''}$$

   Which under translation (and the same assumptions about the contexts) is

$$\frac{\begin{array}{c}[\![\Gamma]\!] \mid [\![\Delta']\!] \vdash [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]]] : [\![\underline{B}]\!] \\ [\![\Gamma']\!] \mid [\![\Delta'']\!] \vdash [\![M']\!][[\![\Psi']\!]] \sqsubseteq [\![\langle \underline{B}' \swarrow \underline{B}''\rangle]\!][[\![M'']\!][[\![\Phi']\!]]] : [\![\underline{B}']\!]\end{array}}{[\![\Gamma]\!] \mid [\![\Delta'']\!] \vdash [\![M]\!][[\![\Psi'']\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}''\rangle]\!][[\![M'']\!][[\![\Phi'']\!]]] : [\![\underline{B}]\!]}$$

We proceed as follows, the key lemma here is the cast decomposition lemma:

$$\begin{aligned}
[\![M]\!][[\![\Psi'']\!]] &\sqsupseteq\sqsubseteq [\![M]\!][[\![\Psi]\!]][[\![\Psi']\!]] && \text{(Cast decomposition)}\\
&\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Psi']\!]][[\![\Phi]\!]]] && \text{(IH)}\\
&\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![\langle \underline{B}' \swarrow \underline{B}''\rangle]\!][[\![M'']\!][[\![\Phi']\!]][[\![\Phi]\!]]]] && \text{(IH)}\\
&\sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}''\rangle]\!][[\![M'']\!][[\![\Phi'']\!]] && \text{(Cast decomposition)}
\end{aligned}$$

3. Substitution of a value in a value: The GTT rule is

$$\frac{\Phi, x \sqsubseteq x' : A_1 \sqsubseteq A_1' \vdash V_2 \sqsubseteq V_2' : A_2 \sqsubseteq A_2' \qquad \Phi \vdash V_1 \sqsubseteq V_1' : A_1 \sqsubseteq A_1'}{\Phi \vdash V_2[V_1/x] \sqsubseteq V_2'[V_1'/x'] : A_2 \sqsubseteq A_2'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$. Under translation, we need to show

$$\frac{[\![\Gamma]\!], x : [\![A_1]\!] \vdash [\![\langle A_2' \nwarrow A_2\rangle]\!][[\![V_2]\!]] \sqsubseteq [\![V_2']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1\rangle]\!][x]/x'] : [\![A_2']\!] \qquad [\![\Gamma]\!] \vdash [\![\langle A_1' \nwarrow A_1\rangle]\!][[\![V_1]\!]] \sqsubseteq [\![V_1']\!][[\![\Phi]\!]] : [\![A_1']\!]}{[\![\Gamma]\!] \vdash [\![\langle A_2' \nwarrow A_2\rangle]\!][[\![V_2[V_1/x]]\!]] \sqsubseteq [\![V_2'[V_1'/x']]\!][[\![\Phi]\!]] : [\![A_2']\!]}$$

Which follows by compositionality:

$$\begin{aligned}
[\![\langle A_2' \nwarrow A_2\rangle]\!][[\![V_2[V_1/x]]\!]] &= ([\![\langle A_2' \nwarrow A_2\rangle]\!][[\![V_2]\!]])[[\![V_1]\!]/x] && \text{(Compositionality)}\\
&\sqsubseteq [\![V_2']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1\rangle]\!][x]/x'][[\![V_1]\!]/x] && \text{(IH)}\\
&= [\![V_2']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1\rangle]\!][[\![V_1]\!]]/x']\\
&\sqsubseteq [\![V_2']\!][[\![\Phi]\!]][[\![V_1']\!][[\![\Phi]\!]]/x'] && \text{(IH)}\\
&= [\![V_2'[V_1'/x']]\!][[\![\Phi]\!]]
\end{aligned}$$

4. Substitution of a value in a term: The GTT rule is

$$\frac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \mid \Psi \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}' \qquad \Phi \vdash V \sqsubseteq V' : A \sqsubseteq A'}{\Phi \vdash M[V/x] \sqsubseteq M'[V'/x'] : \underline{B} \sqsubseteq \underline{B}'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$ and $\Psi : \Delta \sqsubseteq \Delta'$. Under translation this is:

$$\frac{[\![\Gamma]\!], x : [\![A]\!] \mid [\![\Delta]\!] \vdash [\![M]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A' \nwarrow A\rangle]\!][x]/x']] : [\![\underline{B}]\!] \qquad [\![\Gamma]\!] \vdash [\![\langle A' \nwarrow A\rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] : [\![A']\!]}{[\![\Gamma]\!] \mid [\![\Delta]\!] \vdash [\![M[V/x]]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M'[V'/x']]\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]}$$

Which follows from compositionality of the translation:

$$\begin{aligned}
[\![M[V/x]]\!] &= [\![M]\!][[\![V]\!]/x] && \text{(Compositionality)}\\
&\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A' \nwarrow A \rangle]\!][x/x']][[\![V]\!]/x] && \text{(IH)}\\
&= [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A' \nwarrow A \rangle]\!][[\![V]\!]/x']]\\
&\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![V']\!][[\![\Phi]\!]]/x']] && \text{(IH)}\\
&= [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M'[V'/x']]\!][[\![\Phi]\!]]] && \text{(Compositionality)}
\end{aligned}$$

5. Substitution of a term in a stack: The GTT rule is

$$\frac{\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash S \sqsubseteq S' : \underline{C} \sqsubseteq \underline{C}' \qquad \Phi \mid \cdot \vdash M \sqsubseteq M' : \underline{B} \sqsubseteq \underline{B}'}{\Phi \mid \cdot \vdash S[M] \sqsubseteq S'[M'] : \underline{C} \sqsubseteq \underline{C}'}$$

Where $\Phi : \Gamma \sqsubseteq \Gamma'$. Under translation this is

$$\frac{[\![\Gamma]\!] \mid \bullet : [\![\underline{B}']\!] \vdash [\![S]\!][[\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][\bullet]] \sqsubseteq [\![\langle \underline{C} \swarrow \underline{C}' \rangle]\!][[\![S']\!][[\![\Phi]\!]]] : [\![\underline{C}]\!] \qquad [\![\Gamma]\!] \mid \cdot \vdash [\![M]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]] : [\![\underline{B}]\!]}{[\![\Gamma]\!] \mid \cdot \vdash [\![S[M]]\!] \sqsubseteq [\![\langle \underline{C} \swarrow \underline{C}' \rangle]\!][[\![S'[M']]\!][[\![\Phi]\!]]] : [\![\underline{C}]\!]}$$

We follows easily using compositionality of the translation:

$$\begin{aligned}
[\![S[M]]\!] &= [\![S]\!][[\![M]\!]] && \text{(Compositionality)}\\
&\sqsubseteq [\![S]\!][[\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]] && \text{(IH)}\\
&\sqsubseteq [\![\langle \underline{C} \swarrow \underline{C}' \rangle]\!][[\![S']\!][[\![\Phi]\!]][[\![M']\!][[\![\Phi]\!]]] && \text{(IH)}\\
&= [\![\langle \underline{C} \swarrow \underline{C}' \rangle]\!][[\![S'[M']]\!][[\![\Phi]\!]] && \text{(Compositionality)}
\end{aligned}$$

6. Variables: The GTT rule is

$$\Gamma_1 \sqsubseteq \Gamma_1', x \sqsubseteq x' : A \sqsubseteq A', \Gamma_2 \sqsubseteq \Gamma_2' \vdash x \sqsubseteq x' : A \sqsubseteq A'$$

which under translation is

$$[\![\Gamma_1]\!], x : [\![A]\!], [\![\Gamma_2]\!] \vdash [\![\langle A' \nwarrow A \rangle]\!][x] \sqsubseteq [\![\langle A' \nwarrow A \rangle]\!][x] : [\![A']\!]$$

which is an instance of reflexivity.

7. Hole: The GTT rule is

$$\Phi \mid \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$$

which under translation is

$$[\![\Gamma]\!] \mid \bullet : \underline{B}' \vdash [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][\bullet] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][\bullet] : \underline{B}$$

which is an instance of reflexivity.

8. Error is bottom: The GTT axiom is

   $$\Phi \vdash \mho \sqsubseteq M : \underline{B}$$

   where $\Phi : \Gamma \sqsubseteq \Gamma'$, so we need to show

   $$[\![\Gamma]\!] \vdash \mho \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B} \rangle]\!][[\![M]\!][[\![\Phi]\!]] : [\![\underline{B}]\!]$$

   which is an instance of the error is bottom axiom of CBPV.

9. Error strictness: The GTT axiom is

   $$\Phi \vdash S[\mho] \sqsubseteq \mho : \underline{B}$$

   where $\Phi : \Gamma \sqsubseteq \Gamma'$, which under translation is

   $$[\![\Gamma]\!] \vdash [\![S]\!][\mho] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B} \rangle]\!][\mho] : [\![\underline{B}]\!]$$

   By strictness of stacks in CBPV, both sides are equivalent to $\mho$, so it follows by reflexivity.

10. UpCast-L: The GTT axiom is

    $$x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \nwarrow A \rangle x \sqsubseteq x' : A'$$

    which under translation is

    $$x : [\![A]\!] \vdash [\![\langle A' \nwarrow A' \rangle]\!][[\![\langle A' \nwarrow A \rangle]\!][x]] \sqsubseteq [\![\langle A' \nwarrow A \rangle]\!][x] : A'$$

    Which follows by identity expansion and reflexivity.

11. UpCast-R: The GTT axiom is

    $$x : A \vdash x \sqsubseteq \langle A' \nwarrow A \rangle x : A \sqsubseteq A'$$

    which under translation is

    $$x : [\![A]\!] \vdash [\![\langle A' \nwarrow A \rangle]\!][x] \sqsubseteq [\![\langle A' \nwarrow A \rangle]\!][[\![\langle A \nwarrow A \rangle]\!][x]] : [\![A']\!]$$

    which follows by identity expansion and reflexivity.

12. DnCast-R: The GTT axiom is

    $$\bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}' \vdash \bullet \sqsubseteq \langle \underline{B} \swarrow \underline{B}' \rangle : \underline{B}$$

    Which under translation is

    $$\bullet : [\![\underline{B}']\!] \vdash [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][\bullet] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B} \rangle]\!][[\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][\bullet]] : [\![\underline{B}]\!]$$

    Which follows by identity expansion and reflexivity.

13. DnCast-L: The GTT axiom is

$$\bullet : \underline{B}' \vdash \langle \underline{B} \nleftarrow \underline{B}' \rangle \bullet \sqsubseteq \bullet : \underline{B} \sqsubseteq \underline{B}'$$

So under translation we need to show

$$\bullet : [\![\underline{B}']\!] \vdash [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][[\![\langle \underline{B}' \nleftarrow \underline{B}' \rangle]\!][\bullet]] \sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!]\bullet : [\![\underline{B}]\!]$$

Which follows immediately by reflexivity and the lemma that identity casts are identities.

14. 0 elim, we do the term case, the value case is similar

$$\frac{\langle 0 \nwarrow 0 \rangle[[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]]}{\mathtt{absurd}\ [\![V]\!] \sqsubseteq \langle \underline{B} \nleftarrow \underline{B}' \rangle \mathtt{absurd}\ [\![V']\!][[\![\Phi]\!]]}$$

Immediate by 0η.

15. $+$ intro, we do the `inl` case, the `inr` case is the same:

$$\frac{[\![\langle A_1' \nwarrow A_1 \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]]}{[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][\mathtt{inl}\ [\![V]\!]] \sqsubseteq \mathtt{inl}\ [\![V']\!][[\![\Phi]\!]]}$$

Which follows easily:

$$\begin{aligned} [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][\mathtt{inl}\ [\![V]\!]] &\sqsupseteq\sqsubseteq \mathtt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!][\![V]\!] \\ &\qquad \text{(cast reduction)} \\ &\sqsubseteq \mathtt{inl}\ [\![V']\!][[\![\Phi]\!]] \qquad \text{(IH)} \end{aligned}$$

16. $+$ elim, we do just the cases where the continuations are terms:

$$\frac{\begin{array}{c}[\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] \\ [\![M_1]\!][[\![\Psi]\!]] \sqsubseteq [\![M_1']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]/x_1'] \\ [\![M_2]\!][[\![\Psi]\!]] \sqsubseteq [\![M_2']\!][[\![\Phi]\!]][[\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]/x_2']\end{array}}{\mathtt{case}\ [\![V]\!]\{x_1.[\![M_1]\!][[\![\Psi]\!]] \mid x_2.[\![M_2]\!][[\![\Psi]\!]]\} \sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![V]\!]'[[\![\Phi]\!]]\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\}]}$$

$$\begin{aligned} &\mathtt{case}\ [\![V]\!]\{x_1.[\![M_1]\!][[\![\Psi]\!]] \mid x_2.[\![M_2]\!][[\![\Psi]\!]]\} \\ &\sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![V]\!]\{x_1.[\![M_1']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]/x_1'] \mid x_2.[\![M_2']\!][[\![\Phi]\!]][[\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]/x_2']\}] \\ &\qquad \text{(IH)} \\ &\sqsupseteq\sqsubseteq \mathtt{case}\ [\![V]\!] \qquad \text{(comm conv)} \\ &\quad \{x_1.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][[\![M_1']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1 \rangle]\!][x_1]/x_1']] \\ &\quad \mid x_2.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][[\![M_2']\!][[\![\Phi]\!]][[\![\langle A_2' \nwarrow A_2 \rangle]\!][x_2]/x_2']]\} \\ &\sqsupseteq\sqsubseteq \mathtt{case}\ [\![V]\!] \qquad (+\beta) \\ &\quad \{x_1.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ \mathtt{inl}\ [\![\langle A_1' \nwarrow A_1 \rangle]\!]x_1\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\} \\ &\quad \mid x_2.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ \mathtt{inr}\ [\![\langle A_2' \nwarrow A_2 \rangle]\!]x_2\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\}]\} \\ &\sqsupseteq\sqsubseteq \mathtt{case}\ [\![V]\!] \qquad \text{(cast reduction)} \\ &\quad \{x_1.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!]\mathtt{inl}\ x_1\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\} \\ &\quad \mid x_2.[\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!]\mathtt{inr}\ x_2\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\}]\} \\ &\sqsupseteq\sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![\langle A_1' + A_2' \nwarrow A_1 + A_2 \rangle]\!][[\![V]\!]\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\}] \\ &\sqsubseteq [\![\langle \underline{B} \nleftarrow \underline{B}' \rangle]\!][\mathtt{case}\ [\![V']\!][[\![\Phi]\!]]\{x_1'.[\![M_1']\!][[\![\Phi]\!]] \mid x_2'.[\![M_2']\!][[\![\Phi]\!]]\}] \qquad \text{(IH)} \end{aligned}$$

17. 1 intro:

$$[\![\langle 1 \nwarrow 1\rangle]\!][()] \sqsubseteq ()$$

Immediate by cast reduction.

18. 1 elim (continuations are terms case):

$$\frac{[\![\langle 1 \nwarrow 1\rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] \quad [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]]}{\mathtt{split}\ [\![V]\!]\ \mathtt{to}\ ().[\![M]\!][[\![\Psi]\!]] \sqsubseteq \langle \underline{B} \swarrow \underline{B}'\rangle[\mathtt{split}\ [\![V]\!]'[[\![\Phi]\!]]\ \mathtt{to}\ ().[\![M']\!][[\![\Phi]\!]]]}$$

which follows by identity expansion Lemma 134.

19. $\times$ intro:

$$\frac{[\![\langle A_1' \nwarrow A_1\rangle]\!][V_1] \sqsubseteq [\![V_1'[[\![\Phi]\!]]]\!] \quad [\![\langle A_2' \nwarrow A_2\rangle]\!][V_2] \sqsubseteq [\![V_2'[[\![\Phi]\!]]]\!]}{[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!][([\![V_1]\!], [\![V_2]\!])] \sqsubseteq ([\![V_1'[[\![\Phi]\!]]]\!], [\![V_2'[[\![\Phi]\!]]]\!])}$$

We proceed:

$$\begin{aligned}
&[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!][([\![V_1]\!], [\![V_2]\!])] \\
&\sqsupseteq\sqsubseteq ([\![\langle A_1' \nwarrow A_1\rangle]\!][V_1]\!], [\![\langle A_2' \nwarrow A_2\rangle]\!][V_2]\!]) && \text{(cast reduction)} \\
&\sqsubseteq ([\![V_1'[[\![\Phi]\!]]]\!], [\![V_2'[[\![\Phi]\!]]]\!]) && \text{(IH)}
\end{aligned}$$

20. $\times$ elim: We show the case where the continuations are terms, the value continuations are no different:

$$\frac{[\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]] \quad [\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1\rangle]\!][x]/x'][[\![\langle A_2' \nwarrow A_2\rangle]\!][y]/y']]}{\mathtt{let}\ (x,y) = [\![V]\!]; [\![M]\!][[\![\Psi]\!]] \sqsubseteq \langle \underline{B} \swarrow \underline{B}'\rangle[\mathtt{let}\ (x',y') = [\![V]\!]'[[\![\Phi]\!]]; [\![M']\!][[\![\Phi]\!]]]}$$

We proceed as follows:

$$\begin{aligned}
&\mathtt{let}\ (x,y) = [\![V]\!]; [\![M]\!][[\![\Psi]\!]] \\
&\sqsubseteq \mathtt{let}\ (x,y) = [\![V]\!]; [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A_1' \nwarrow A_1\rangle]\!][x]/x'][[\![\langle A_2' \nwarrow A_2\rangle]\!][y]/y']] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x,y) = [\![V]\!]; && (\times\beta) \\
&\quad \mathtt{let}\ (x',y') = ([\![\langle A_1' \nwarrow A_1\rangle]\!][x], [\![\langle A_2' \nwarrow A_2\rangle]\!][y]); [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]] \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x,y) = [\![V]\!]; && \text{(cast reduction)} \\
&\quad \mathtt{let}\ (x',y') = [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2'\rangle]\!][(x,y)]; [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]] \\
&\sqsupseteq\sqsubseteq \mathtt{let}\ (x',y') = [\![\langle A_1' \times A_2' \nwarrow A_1 \times A_2\rangle]\!][[\![V]\!]]; [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]] && (\times\eta) \\
&\sqsubseteq \mathtt{let}\ (x',y') = [\![V']\!][[\![\Phi]\!]]; [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][[\![M']\!][[\![\Phi]\!]] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}'\rangle]\!][\mathtt{let}\ (x',y') = [\![V']\!][[\![\Phi]\!]]; [\![M']\!][[\![\Phi]\!]]] && \text{(commuting conversion)}
\end{aligned}$$

21. U intro:

$$\frac{[\![M]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]}{[\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!][\mathtt{thunk}\ [\![M]\!]] \sqsubseteq \mathtt{thunk}\ [\![M']\!][[\![\Phi]\!]]}$$

We proceed as follows:

$$\begin{aligned}
[\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!][\mathtt{thunk}\ [\![M]\!]] &\sqsubseteq [\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!][\mathtt{thunk}\ [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]] \\
&\qquad \text{(IH)} \\
&\sqsubseteq \mathtt{thunk}\ [\![M']\!][[\![\Phi]\!]] \\
&\qquad \text{(alt projection)}
\end{aligned}$$

22. U elim:

$$\frac{[\![\langle U\underline{B}' \nwarrow U\underline{B} \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]]}{\mathtt{force}\ [\![V]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!]\mathtt{force}\ [\![V']\!][[\![\Phi]\!]]}$$

By hom-set formulation of adjunction Lemma 136.

23. $\top$ intro:

$$\{\} \sqsubseteq [\![\langle \top \swarrow \top \rangle]\!][\{\}]$$

Immediate by $\top\eta$

24. & intro:

$$\frac{\begin{array}{c}[\![M_1]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B}_1 \swarrow \underline{B}_1' \rangle]\!][[\![M_1']\!][[\![\Phi]\!]]] \\ [\![M_2]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B}_2 \swarrow \underline{B}_2' \rangle]\!][[\![M_2']\!][[\![\Phi]\!]]]\end{array}}{([\![M_1]\!][[\![\Psi]\!]], [\![M_2]\!][[\![\Psi]\!]]) \sqsubseteq [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][([\![M_1']\!][[\![\Phi]\!]], [\![M_2']\!][[\![\Phi]\!]])]}$$

We proceed as follows:

$$\begin{aligned}
&([\![M_1]\!][[\![\Psi]\!]], [\![M_2]\!][[\![\Psi]\!]]) \\
&\sqsubseteq ([\![\langle \underline{B}_1 \swarrow \underline{B}_1' \rangle]\!][[\![M_1']\!][[\![\Phi]\!]]], [\![\langle \underline{B}_2 \swarrow \underline{B}_2' \rangle]\!][[\![M_2']\!][[\![\Phi]\!]]]) \\
&\qquad \text{(IH)} \\
&\sqsupseteq \sqsubseteq \{\pi \mapsto \pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][([\![M_1']\!][[\![\Phi]\!]], [\![M_2']\!][[\![\Phi]\!]])] \\
&\qquad \text{(cast reduction)} \\
&\quad \mid \pi' \mapsto \pi'[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][([\![M_1']\!][[\![\Phi]\!]], [\![M_2']\!][[\![\Phi]\!]])]\} \\
&\sqsupseteq \sqsubseteq [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][([\![M_1']\!][[\![\Phi]\!]], [\![M_2']\!][[\![\Phi]\!]])] \quad (\&\eta)
\end{aligned}$$

25. & elim, we show the π case, π' is symmetric:

$$\frac{[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][[\![M']\!][[\![\Phi]\!]]}{\pi[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B}_1 \swarrow \underline{B}_1' \rangle]\!][\pi[\![M']\!][[\![\Phi]\!]]]}$$

We proceed as follows:

$$\begin{aligned}
\pi[\![M]\!][[\![\Psi]\!]] &\sqsubseteq \pi[\![\langle \underline{B}_1 \,\&\, \underline{B}_2 \swarrow \underline{B}_1' \,\&\, \underline{B}_2' \rangle]\!][[\![M']\!][[\![\Phi]\!]]] \qquad \text{(IH)} \\
&\sqsupseteq \sqsubseteq [\![\langle \underline{B}_1 \swarrow \underline{B}_1' \rangle]\!][\pi[\![M']\!][[\![\Phi]\!]]] \quad \text{(cast reduction)}
\end{aligned}$$

26.

$$\frac{[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A' \nwarrow A \rangle]\!]x/x']]}{\lambda x : A.[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle]\!][\lambda x' : A'.[\![M']\!][[\![\Phi]\!]]]}$$

We proceed as follows:

$$\begin{aligned}
&\lambda x : A.[\![M]\!][[\![\Psi]\!]] \\
&\sqsubseteq \lambda x : A.[\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]][[\![\langle A' \nwarrow A \rangle]\!]x/x']] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq \lambda x : A.([\![\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle]\!][\lambda x'.[\![M']\!][[\![\Phi]\!]])\, x \\
& && \text{(cast reduction)} \\
&\sqsupseteq\sqsubseteq [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle]\!][\lambda x'.[\![M']\!][[\![\Phi]\!]] && (\to \eta)
\end{aligned}$$

27. We need to show

$$\frac{\begin{array}{c}[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]] \\ [\![\langle A' \nwarrow A \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]]\end{array}}{[\![M]\!][[\![\Psi]\!]]\, [\![V]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]\, [\![V']\!][[\![\Phi]\!]]]}$$

We proceed:

$$\begin{aligned}
&[\![M]\!][[\![\Psi]\!]]\, [\![V]\!] \\
&\sqsubseteq ([\![\langle A \to \underline{B} \swarrow A' \to \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]])\, [\![V]\!] && \text{(IH)} \\
&\sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]\, ([\![\langle A' \nwarrow A \rangle]\!][[\![V]\!])] \\
& && \text{(cast reduction)} \\
&\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![M']\!][[\![\Phi]\!]]\, [\![V']\!][[\![\Phi]\!]]] && \text{(IH)}
\end{aligned}$$

28. We need to show

$$\frac{[\![\langle A' \nwarrow A \rangle]\!][[\![V]\!]] \sqsubseteq [\![V']\!][[\![\Phi]\!]]}{\mathtt{ret}\ [\![V]\!] \sqsubseteq [\![\langle \underline{F}A \swarrow \underline{F}A' \rangle]\!][\mathtt{ret}\ [\![V']\!][[\![\Phi]\!]]]}$$

By hom-set definition of adjunction Lemma 136

29. We need to show

$$\frac{\begin{array}{c}[\![M]\!][[\![\Psi]\!]] \sqsubseteq [\![\langle \underline{F}A \swarrow \underline{F}A' \rangle]\!][[\![M']\!][\Phi]] \\ [\![N]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![N]\!][\Phi][[\![\langle A' \nwarrow A \rangle]\!]x/x']]\end{array}}{x \leftarrow [\![M]\!][[\![\Psi]\!]]; [\![N]\!] \sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][x' \leftarrow [\![M']\!][[\![\Phi]\!]]; [\![N']\!][[\![\Phi]\!]]]}$$

We proceed:

$$\begin{aligned}
&x \leftarrow [\![M]\!][[\![\Psi]\!]]; [\![N]\!] \\
&\sqsubseteq x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A' \rangle]\!][[\![M']\!][\Phi]]; [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![N]\!][\Phi][[\![\langle A' \nwarrow A \rangle]\!]x/x']] \\
& && \text{(IH, congruence)} \\
&\sqsupseteq\sqsubseteq x \leftarrow [\![\langle \underline{F}A \swarrow \underline{F}A' \rangle]\!][[\![M']\!][\Phi]]; \\
&\quad x' \leftarrow \mathtt{ret}\ [\![\langle A' \nwarrow A \rangle]\!][x]; [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![N]\!][\Phi]] && (\underline{F}\beta) \\
&\sqsubseteq x' \leftarrow [\![M']\!][\Phi]; [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][[\![N]\!][\Phi]] && \text{(Projection)} \\
&\sqsupseteq\sqsubseteq [\![\langle \underline{B} \swarrow \underline{B}' \rangle]\!][x' \leftarrow [\![M']\!][\Phi]; [\![N]\!][\Phi]] \\
& && \text{(commuting conversion)}
\end{aligned}$$

$$
\begin{array}{lcl}
A & ::= & X \mid \mu X.A \mid U\underline{B} \mid 0 \mid A_1 + A_2 \mid 1 \mid A_1 \times A_2 \\
\underline{B} & ::= & \underline{Y} \mid \nu \underline{Y}.\underline{B} \mid \underline{F}A \mid \top \mid \underline{B}_1 \,\&\, \underline{B}_2 \mid A \to \underline{B} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \\
\Delta & ::= & \cdot \mid \bullet : \underline{B} \\
V & ::= & x \mid \mathtt{roll}_{\mu X.A}\ V \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V \mid () \mid (V_1, V_2) \mid \mathtt{thunk}\ M \\
M & ::= & \mathcal{U}_{\underline{B}} \mid \mathtt{let}\ x = V; M \mid \mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.M \mid \mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M \mid \mathtt{unroll}\ M \mid \mathtt{abort}\ V \mid \\
 & & \mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \mid \mathtt{split}\ V\ \mathtt{to}\ ().M \mid \mathtt{let}\ (x,y) = V; M \mid \mathtt{force}\ V \mid \\
 & & \mathtt{ret}\ V \mid x \leftarrow M; N \mid \lambda x : A.M \mid M\,V \mid \{\} \mid (M_1, M_2) \mid \pi M \mid \pi' M \\
S & ::= & \bullet \mid x \leftarrow S; M \mid S\,V \mid \pi S \mid \pi' S \mid \mathtt{unroll}_{\nu \underline{Y}.\underline{B}}\ S
\end{array}
$$

Figure 7.7: Operational CBPV Syntax

□

As a corollary, we have the following conservativity result, which says that the homogeneous term precisions in GTT are sound and complete for inequalities in CBPV*.

Corollary 137 (Conservativity)**.** *If $\Gamma \mid \Delta \vdash E, E' : T$ are two terms of the same type in the intersection of GTT and CBPV*, then $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ is provable in GTT iff it is provable in CBPV*.*

Proof. The reverse direction holds because CBPV* is a syntactic subset of GTT. The forward direction holds by axiomatic graduality and the fact that identity casts are identities. □

7.3 COMPLEX VALUE/STACK ELIMINATION

Next, to bridge the gap between the semantic notion of complex value and stack with the more rigid operational notion, we perform a complexity-elimination pass. This translates a computation with complex values in it to an equivalent computation without complex values: i.e., all pattern matches take place in computations, rather than in values, and translates a term precision derivation that uses complex stacks to one that uses only "simple" stacks without pattern-matching and computation introduction forms. Stacks do not appear anywhere in the grammar of terms, but they are used in the equational theory (computation η rules and error strictness). This translation clarifies the behavioral meaning of complex values and stacks, following Führmann [27] and Munch-Maccagnoni [53], and therefore of upcasts and downcasts.

The syntax of operational CBPV is as in Figure 5.1 (unshaded), but with recursive types added as in §7.1, and with values and stacks restricted as in Figure 7.7.

In CBPV, values include only introduction forms, as usual for values in operational semantics, and CBPV stacks consist only of elimination

forms for computation types (the syntax of CBPV enforces an A-normal form, where only values can be pattern-matched on, so `case` and `split` are not evaluation contexts in the operational semantics).

Levy [45] translates CBPV* to CBPV, but not does not prove the inequality preservation that we require here, so we give an alternative translation for which this property is easy to verify. We translate both complex values and complex stacks to fully general computations, so that computation pattern-matching can replace the pattern-matching in complex values/stacks. For example, for a closed value, we could "evaluate away" the complexity and get a closed simple value (if we don't use U), but for open terms, evaluation will get "stuck" if we pattern match on a variable—so not every complex value can be translated to a value in CBPV. More formally, we translate a CBPV* complex value $V : A$ to a CBPV computation $V^\dagger : \underline{F}A$ that in CBPV* is equivalent to $\mathtt{ret}\ V$. Similarly, we translate a CBPV* complex stack S with hole $\bullet : \underline{B}$ to a CBPV computation $S^\dagger$ with a free variable $z : U\underline{B}$ such that in CBPV*, $S^\dagger \sqsupseteq\sqsubseteq S[\mathtt{force}\ z]$. Computations $M : \underline{B}$ are translated to computations $M^\dagger$ with the same type.

The *de-complexification* procedure is defined as follows.

Definition 138 (De-complexification)**.** We define

$$
\begin{array}{rcl}
\bullet^\dagger & = & \mathtt{force}\ z \\
x^\dagger & = & \mathtt{ret}\ x \\
\\
(\mathtt{ret}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{ret}\ x \\
(M\ V)^\dagger & = & x \leftarrow V^\dagger; M^\dagger\ x \\
\\
(\mathtt{force}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{force}\ x \\
(\mathtt{absurd}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{absurd}\ x \\
(\mathtt{case}\ V\{x_1.E_1 \mid x_2.E_2\})^\dagger & = & x \leftarrow V^\dagger; \mathtt{case}\ x\{x_1.{E_1}^\dagger \mid x_2.{E_2}^\dagger\} \\
(\mathtt{split}\ V\ \mathtt{to}\ ().E)^\dagger & = & w \leftarrow V; \mathtt{split}\ w\ \mathtt{to}\ ().E^\dagger \\
(\mathtt{let}\ (x,y) = V; E)^\dagger & = & w \leftarrow V; \mathtt{let}\ (x,y) = w; E^\dagger \\
(\mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.E)^\dagger & = & y \leftarrow V^\dagger; \mathtt{unroll}\ y\ \mathtt{to}\ \mathtt{roll}\ x.E^\dagger \\
\\
(\mathtt{inl}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{ret}\ \mathtt{inl}\ x \\
(\mathtt{inr}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{ret}\ \mathtt{inr}\ x \\
()^\dagger & = & \mathtt{ret}\ () \\
(V_1, V_2)^\dagger & = & x_1 \leftarrow {V_1}^\dagger; x_2 \leftarrow {V_2}^\dagger; \mathtt{ret}\ (x_1, x_2) \\
(\mathtt{thunk}\ M)^\dagger & = & \mathtt{ret}\ \mathtt{thunk}\ M^\dagger \\
(\mathtt{roll}\ V)^\dagger & = & x \leftarrow V^\dagger; \mathtt{roll}\ x
\end{array}
$$

The translation is type-preserving and the identity from CBPV*'s point of view

Lemma 139 (De-complexification De-complexifies)**.** *For any CBPV* term $\Gamma \mid \Delta \vdash E : T$, $E^\dagger$ is a term of CBPV satisfying $\Gamma, \Delta^\dagger \vdash E^\dagger : T^\dagger$ where $\cdot^\dagger = \cdot$ $(\bullet : \underline{B})^\dagger = z : U\underline{B}$, $\underline{B}^\dagger = \underline{B}$, $A^\dagger = \underline{F}A$.*

$$\Gamma, x : A, \Gamma' \vdash x \sqsubseteq x : A \qquad \Gamma \mid \bullet : \underline{B} \vdash \bullet \sqsubseteq \bullet : \underline{B} \qquad \Gamma \vdash \mho \sqsubseteq \mho : \underline{B}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A \qquad \Gamma, x : A \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \mathtt{let}\ x = V; M \sqsubseteq \mathtt{let}\ x = V'; M' : \underline{B}} \qquad \frac{\Gamma \vdash V \sqsubseteq V' : 0}{\Gamma \vdash \mathtt{abort}\ V \sqsubseteq \mathtt{abort}\ V' : \underline{B}}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A_1}{\Gamma \vdash \mathtt{inl}\ V \sqsubseteq \mathtt{inl}\ V' : A_1 + A_2} \qquad \frac{\Gamma \vdash V \sqsubseteq V' : A_2}{\Gamma \vdash \mathtt{inr}\ V \sqsubseteq \mathtt{inr}\ V' : A_1 + A_2}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash M_1 \sqsubseteq M_1' : \underline{B} \qquad \Gamma, x_2 : A_2 \vdash M_2 \sqsubseteq M_2' : \underline{B}}{\Gamma \vdash \mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \sqsubseteq \mathtt{case}\ V'\{x_1.M_1' \mid x_2.M_2'\} : \underline{B}}$$

$$\Gamma \vdash () \sqsubseteq () : 1 \qquad \frac{\Gamma \vdash V_1 \sqsubseteq V_1' : A_1 \qquad \Gamma \vdash V_2 \sqsubseteq V_2' : A_2}{\Gamma \vdash (V_1, V_2) \sqsubseteq (V_1', V_2') : A_1 \times A_2}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \mathtt{let}\ (x, y)\ =\ V; M \sqsubseteq \mathtt{let}\ (x, y)\ =\ V'; M' : \underline{B}}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A[\mu X.A/X]}{\Gamma \vdash \mathtt{roll}_{\mu X.A}\ V \sqsubseteq \mathtt{roll}_{\mu X.A}\ V' : \mu X.A}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.M \sqsubseteq \mathtt{unroll}\ V'\ \mathtt{to}\ \mathtt{roll}\ x.M' : \underline{B}}$$

$$\frac{\Gamma \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \mathtt{thunk}\ M \sqsubseteq \mathtt{thunk}\ M' : U\underline{B}} \qquad \frac{\Gamma \vdash V \sqsubseteq V' : U\underline{B}}{\Gamma \vdash \mathtt{force}\ V \sqsubseteq \mathtt{force}\ V' : \underline{B}}$$

$$\frac{\Gamma \vdash V \sqsubseteq V' : A}{\Gamma \vdash \mathtt{ret}\ V \sqsubseteq \mathtt{ret}\ V' : \underline{F}A} \qquad \frac{\Gamma \vdash M \sqsubseteq M' : \underline{F}A \qquad \Gamma, x : A \vdash N \sqsubseteq N' : \underline{B}}{\Gamma \vdash x \leftarrow M; N \sqsubseteq x \leftarrow M'; N' : \underline{B}}$$

$$\frac{\Gamma, x : A \vdash M \sqsubseteq M' : \underline{B}}{\Gamma \vdash \lambda x : A.M \sqsubseteq \lambda x : A.M' : A \to \underline{B}}$$

$$\frac{\Gamma \vdash M \sqsubseteq M' : A \to \underline{B} \qquad \Gamma \vdash V \sqsubseteq V' : A}{\Gamma \vdash M\,V \sqsubseteq M'\,V' : \underline{B}}$$

$$\frac{\Gamma \vdash M_1 \sqsubseteq M_1' : \underline{B}_1 \qquad \Gamma \vdash M_2 \sqsubseteq M_2' : \underline{B}_2}{\Gamma \vdash (M_1, M_2) \sqsubseteq (M_1', M_2') : \underline{B}_1 \,\&\, \underline{B}_2} \qquad \frac{\Gamma \vdash M \sqsubseteq M' : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vdash \pi M \sqsubseteq \pi M' : \underline{B}_1}$$

$$\frac{\Gamma \vdash M \sqsubseteq M' : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vdash \pi' M \sqsubseteq \pi' M' : \underline{B}_2} \qquad \frac{\Gamma \vdash M \sqsubseteq M' : \underline{B}[\nu\underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \vdash \mathtt{roll}_{\nu\underline{Y}.\underline{B}}\ M \sqsubseteq \mathtt{roll}_{\nu\underline{Y}.\underline{B}}\ M' : \nu\underline{Y}.\underline{B}}$$

$$\frac{\Gamma \vdash M \sqsubseteq M' : \nu\underline{Y}.\underline{B}}{\Gamma \vdash \mathtt{unroll}\ M \sqsubseteq \mathtt{unroll}\ M' : \underline{B}[\nu\underline{Y}.\underline{B}/\underline{Y}]}$$

Figure 7.8: CBPV Inequational Theory (Congruence Rules)

$$\texttt{case inl}\ V\{x_1.M_1 \mid x_2.M_2\} \sqsupseteq\sqsubseteq M_1[V/x_1]$$

$$\texttt{case inr}\ V\{x_1.M_1 \mid x_2.M_2\} \sqsupseteq\sqsubseteq M_2[V/x_2]$$

$$\frac{\Gamma, x : A_1 + A_2 \vdash M : \underline{B}}{\Gamma, x : A_1 + A_2 \vdash M \sqsupseteq\sqsubseteq \texttt{case}\ x\{x_1.M[\texttt{inl}\ x_1/x] \mid x_2.M[\texttt{inr}\ x_2/x]\} : \underline{B}}$$

$$\texttt{let}\ (x_1, x_2) = (V_1, V_2); M \sqsupseteq\sqsubseteq M[V_1/x_1, V_2/x_2]$$

$$\frac{\Gamma, x : A_1 \times A_2 \vdash M : \underline{B}}{\Gamma, x : A_1 \times A_2 \vdash M \sqsupseteq\sqsubseteq \texttt{let}\ (x_1, x_2) = x; M[(x_1, x_2)/x] : \underline{B}}$$

$$\frac{\Gamma, x : 1 \vdash M : \underline{B}}{\Gamma, x : 1 \vdash M \sqsupseteq\sqsubseteq M[()/x] : \underline{B}} \qquad \texttt{unroll roll}_A\ V\ \texttt{to roll}\ x.M \sqsupseteq\sqsubseteq M[V/x]$$

$$\frac{\Gamma, x : \mu X.A \vdash M : \underline{B}}{\Gamma, x : \mu X.A \vdash M \sqsupseteq\sqsubseteq \texttt{unroll}\ x\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x] : \underline{B}}$$

$$\texttt{force thunk}\ M \sqsupseteq\sqsubseteq M \qquad \frac{\Gamma \vdash V : U\underline{B}}{\Gamma \vdash V \sqsupseteq\sqsubseteq \texttt{thunk force}\ V : U\underline{B}}$$

$$\texttt{let}\ x = V; M \sqsupseteq\sqsubseteq M[V/x] \qquad x \leftarrow \texttt{ret}\ V; M \sqsupseteq\sqsubseteq M[V/x]$$

$$\Gamma \mid \bullet : \underline{F}A \vdash \bullet \sqsupseteq\sqsubseteq x \leftarrow \bullet; \texttt{ret}\ x : \underline{F}A \qquad (\lambda x : A.M)\ V \sqsupseteq\sqsubseteq M[V/x]$$

$$\frac{\Gamma \vdash M : A \to \underline{B}}{\Gamma \vdash M \sqsupseteq\sqsubseteq \lambda x : A.M\, x : A \to \underline{B}} \qquad \pi(M, M') \sqsupseteq\sqsubseteq M \qquad \pi'(M, M') \sqsupseteq\sqsubseteq M'$$

$$\frac{\Gamma \vdash M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vdash M \sqsupseteq\sqsubseteq (\pi M, \pi' M) : \underline{B}_1 \,\&\, \underline{B}_2} \qquad \frac{\Gamma \vdash M : \top}{\Gamma \vdash M \sqsupseteq\sqsubseteq \{\} : \top}$$

$$\texttt{unroll roll}_{\underline{B}}\ M \sqsupseteq\sqsubseteq M \qquad \frac{\Gamma \vdash M : \nu\underline{Y}.\underline{B}}{\Gamma \vdash M \sqsupseteq\sqsubseteq \texttt{roll}_{\nu\underline{Y}.\underline{B}}\ \texttt{unroll}\ M : \nu\underline{Y}.\underline{B}}$$

Figure 7.9: CBPV β, η rules

$$\Gamma \vdash \mho \sqsubseteq M : \underline{B} \qquad \Gamma \vdash S[\mho] \sqsupseteq \sqsubseteq \mho : \underline{B} \qquad \Gamma \vdash M \sqsubseteq M : \underline{B} \qquad \Gamma \vdash V \sqsubseteq V : A$$

$$\Gamma \mid \underline{B} \vdash S \sqsubseteq S : \underline{B}' \qquad \frac{\Gamma \vdash M_1 \sqsubseteq M_2 : \underline{B} \qquad \Gamma \vdash M_2 \sqsubseteq M_3 : \underline{B}}{\Gamma \vdash M_1 \sqsubseteq M_3 : \underline{B}}$$

$$\frac{\Gamma \vdash V_1 \sqsubseteq V_2 : A \qquad \Gamma \vdash V_2 \sqsubseteq V_3 : A}{\Gamma \vdash V_1 \sqsubseteq V_3 : A}$$

$$\frac{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \mid \underline{B} \vdash S_2 \sqsubseteq S_3 : \underline{B}'}{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_3 : \underline{B}'}$$

$$\frac{\Gamma, x : A \vdash M_1 \sqsubseteq M_2 : \underline{B} \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \vdash M_1[V_1/x] \sqsubseteq M_2[V_2/x] : \underline{B}}$$

$$\frac{\Gamma, x : A \vdash V_1' \sqsubseteq V_2' : A' \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \vdash V_1'[V_1/x] \sqsubseteq V_2'[V_2/x] : A'}$$

$$\frac{\Gamma, x : A \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \vdash V_1 \sqsubseteq V_2 : A}{\Gamma \mid \underline{B} \vdash S_1[V_1/x] \sqsubseteq S_2[V_2/x] : \underline{B}'}$$

$$\frac{\Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}' \qquad \Gamma \vdash M_1 \sqsubseteq M_2 : \underline{B}}{\Gamma \vdash S_1[M_1] \sqsubseteq S_2[M_2] : \underline{B}'}$$

$$\frac{\Gamma \mid \underline{B}' \vdash S_1' \sqsubseteq S_2' : \underline{B}'' \qquad \Gamma \mid \underline{B} \vdash S_1 \sqsubseteq S_2 : \underline{B}'}{\Gamma \mid \underline{B} \vdash S_1'[S_1] \sqsubseteq S_2'[S_2] : \underline{B}''}$$

Figure 7.10: CBPV logical and error rules

Lemma 140 (De-complexification is Identity in CBPV*)**.** *Considering CBPV as a subset of CBPV* we have*

1. *If* $\Gamma \mid \cdot \vdash M : \underline{B}$ *then* $M \sqsupseteq\sqsubseteq M^\dagger$.
2. *If* $\Gamma \mid \Delta \vdash S : \underline{B}$ *then* $S[\texttt{force}\ z] \sqsupseteq\sqsubseteq S^\dagger$.
3. *If* $\Gamma \vdash V : A$ *then* $\texttt{ret}\ V \sqsupseteq\sqsubseteq V^\dagger$.

Furthermore, if M, V, S *are in CBPV, the proof holds in CBPV.*

Finally, we need to show that the translation preserves inequalities ($E^\dagger \sqsubseteq E'^\dagger$ if $E \sqsubseteq E'$), but because complex values and stacks satisfy more equations than arbitrary computations in the types of their translations do, we need to isolate the special "purity" property that their translations have. We show that complex values are translated to computations that satisfy *thunkability* [53], which intuitively means M should have no observable effects, and so can be freely duplicated or discarded like a value. In the inequational theory of CBPV, this is defined by saying that running M to a value and then duplicating its value is the same as running M every time we need its value:

Definition 141 (Thunkable Computation)**.** A computation $\Gamma \vdash M : \underline{F}A$ is *thunkable* if

$$\Gamma \vdash \texttt{ret}\ (\texttt{thunk}\ M) \sqsupseteq\sqsubseteq x \leftarrow M; \texttt{ret}\ (\texttt{thunk}\ (\texttt{ret}\ x)) : \underline{F}U\underline{F}A$$

Dually, we show that complex stacks are translated to computations that satisfy (semantic) *linearity* [53], where intuitively a computation M with a free variable $x : U\underline{B}$ is linear in x if M behaves as if when it is forced, the first thing it does is forces x, and that is the only time it uses x. This is described in the CBPV inequational theory as follows: if we have a thunk $z : U\underline{F}U\underline{B}$, then either we can force it now and pass the result to M as x, or we can just run M with a thunk that will force z each time M is forced—but if M forces x exactly once, first, these two are the same.

Definition 142 (Linear Term)**.** A term $\Gamma, x : U\underline{B} \vdash M : \underline{C}$ is *linear in* x if

$$\Gamma, z : U\underline{F}U\underline{B} \vdash x \leftarrow \texttt{force}\ z; M \sqsupseteq\sqsubseteq M[\texttt{thunk}\ (x \leftarrow (\texttt{force}\ z); \texttt{force}\ x)]$$

Thunkability/linearity of the translations of complex values/stacks are used to prove the preservation of the η principles for positive types and the strictness of complex stacks with respect to errors under decomplexification.

We need a few lemmas about thunkables and linears to prove that complex values become thunkable and complex stacks become linear. Many of them correspond to program optimizations that are valid for thunkable/linear terms and therefore apply to upcasts and downcasts.

First, the following lemma is useful for optimizing programs with thunkable subterms. Intuitively, since a thunkable has "no effects"

it can be reordered past any other effectful binding. Furhmann [27] calls a morphism that has this property *central* (after the center of a group, which is those elements that commute with every element of the whole group).

Lemma 143 (Thunkables are Central)**.** *If* $\Gamma \vdash M : \underline{F}A$ *is thunkable and* $\Gamma \vdash N : \underline{F}A'$ *and* $\Gamma, x : A, y : A' \vdash N' : \underline{B}$*, then*

$$x \leftarrow M; y \leftarrow N; N' \sqsupseteq\sqsubseteq y \leftarrow N; x \leftarrow M; N'$$

Proof.

$$\begin{aligned}
&x \leftarrow M; y \leftarrow N; N' \\
&\sqsupseteq\sqsubseteq x \leftarrow M; y \leftarrow N; x \leftarrow \texttt{force thunk ret}\ x; N' && (U\beta, \underline{F}\beta) \\
&\sqsupseteq\sqsubseteq x \leftarrow M; w \leftarrow \texttt{ret thunk ret}\ x; y \leftarrow N; x \leftarrow \texttt{force}\ w; N' && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq w \leftarrow (x \leftarrow M; \texttt{ret thunk ret}\ x); y \leftarrow N; x \leftarrow \texttt{force}\ w; N' && (\underline{F}\eta) \\
&\sqsupseteq\sqsubseteq w \leftarrow \texttt{ret thunk}\ M; y \leftarrow N; x \leftarrow \texttt{force}\ w; N' && (M \text{ thunkable}) \\
&\sqsupseteq\sqsubseteq y \leftarrow N; x \leftarrow \texttt{force thunk}\ M; N' && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq y \leftarrow N; x \leftarrow M; N' && (U\beta)
\end{aligned}$$

□

Next, we show thunkables are closed under composition and that return of a value is always thunkable. This allows us to easily build up bigger thunkables from smaller ones.

Lemma 144 (Thunkables compose)**.** *If* $\Gamma \vdash M : \underline{F}A$ *and* $\Gamma, x : A \vdash N : \underline{F}A'$ *are thunkable, then*

$$x \leftarrow M; N$$

is thunkable.

Proof.

$$
\begin{aligned}
& y \leftarrow (x \leftarrow M; N); \texttt{ret thunk ret } y \\
& \sqsupseteq\sqsubseteq x \leftarrow M; y \leftarrow N; \texttt{ret thunk ret } y && (\underline{F}\eta) \\
& \sqsupseteq\sqsubseteq x \leftarrow M; \texttt{ret thunk } N && (N \text{ thunkable}) \\
& \sqsupseteq\sqsubseteq x \leftarrow M; \texttt{ret thunk } (x \leftarrow \texttt{ret } x; N) && (\underline{F}\beta) \\
& \sqsupseteq\sqsubseteq x \leftarrow M; w \leftarrow \texttt{ret thunk ret } x; \texttt{ret thunk } (x \leftarrow \texttt{force } w; N) && (\underline{F}\beta, U\beta) \\
& \sqsupseteq\sqsubseteq w \leftarrow (x \leftarrow M; \texttt{ret thunk ret } x); \texttt{ret thunk } (x \leftarrow \texttt{force } w; N) && (\underline{F}\eta) \\
& \sqsupseteq\sqsubseteq w \leftarrow \texttt{ret thunk } M; \texttt{ret thunk } (x \leftarrow \texttt{force } w; N) && (M \text{ thunkable}) \\
& \sqsupseteq\sqsubseteq \texttt{ret thunk } (x \leftarrow \texttt{force thunk } M; N) && (\underline{F}\beta) \\
& \sqsupseteq\sqsubseteq \texttt{ret thunk } (x \leftarrow M; N) && (U\beta)
\end{aligned}
$$

□

Lemma 145 (Return is Thunkable)**.** *If* $\Gamma \vdash V : A$ *then* $\texttt{ret } V$ *is thunkable.*

Proof. By $\underline{F}\beta$:

$$x \leftarrow \texttt{ret } V; \texttt{ret thunk ret } x \sqsupseteq\sqsubseteq \texttt{ret thunk ret } V$$

□

And we can then prove the desired property for complex values:

Lemma 146 (Complex Values Simplify to Thunkable Terms)**.** *If* $\Gamma \vdash V : A$ *is a (possibly) complex value, then* $\Gamma \vdash V^\dagger : \underline{F}A$ *is thunkable.*

Proof. Introduction forms follow from return is thunkable and thunkables compose. For elimination forms it is sufficient to show that when the branches of pattern matching are thunkable, the pattern match is thunkable.

1. x: We need to show $x^\dagger = \texttt{ret } x$ is thunkable, which we proved as a lemma above.
2. 0 elim, we need to show

   $$y \leftarrow \texttt{absurd } V; \texttt{ret thunk ret } y \sqsupseteq\sqsubseteq \texttt{ret thunk absurd } V$$

   but by $\eta 0$ both sides are equivalent to $\texttt{absurd } V$.
3. $+$ elim, we need to show

$$\texttt{ret thunk } (\texttt{case } V\{x_1.M_1 \mid x_2.M_2\}) \sqsupseteq\sqsubseteq y \leftarrow (\texttt{case } V\{x_1.M_1 \mid x_2.M_2\}); \texttt{ret thunk ret } y$$

$$
\begin{array}{ll}
& \texttt{ret thunk}\ (\texttt{case}\ V\{x_1.M_1 \mid x_2.M_2\}) \\
\sqsupseteq\sqsubseteq & \texttt{case}\ V \qquad (+\eta) \\
& \quad \{x_1.\texttt{ret thunk}\ (\texttt{case inl}\ x_1\{x_1.M_1 \mid x_2.M_2\}) \\
& \quad \mid x_2.\texttt{ret thunk}\ (\texttt{case inr}\ x_2\{x_1.M_1 \mid x_2.M_2\})\} \\
\sqsupseteq\sqsubseteq & \texttt{case}\ V \qquad (+\beta) \\
& \quad \{x_1.\texttt{ret thunk}\ M_1 \\
& \quad \mid x_2.\texttt{ret thunk}\ M_2\} \\
\sqsupseteq\sqsubseteq & \texttt{case}\ V \qquad (M_1, M_2\ \text{thunkable}) \\
& \quad \{x_1.y \leftarrow M_1; \texttt{ret thunk ret}\ y \\
& \quad \mid x_2.y \leftarrow M_2; \texttt{ret thunk ret}\ y\} \\
\sqsupseteq\sqsubseteq & y \leftarrow (\texttt{case}\ V\{x_1.M_1 \mid x_2.M_2\}); \texttt{ret thunk ret}\ y \\
& \qquad \text{(commuting conversion)}
\end{array}
$$

4. $\times$ elim

$$
\begin{array}{ll}
& \texttt{ret thunk}\ (\texttt{let}\ (x,y) = V; M) \\
\sqsupseteq\sqsubseteq & \texttt{let}\ (x,y) = V; \texttt{ret thunk let}\ (x,y) = (x,y); M \\
& \qquad (\times\eta) \\
\sqsupseteq\sqsubseteq & \texttt{let}\ (x,y) = V; \texttt{ret thunk}\ M \qquad (\times\beta) \\
\sqsupseteq\sqsubseteq & \texttt{let}\ (x,y) = V; z \leftarrow M; \texttt{ret thunk ret}\ z \\
& \qquad (M\ \text{thunkable}) \\
\sqsupseteq\sqsubseteq & z \leftarrow (\texttt{let}\ (x,y) = V; M); \texttt{ret thunk ret}\ z \\
& \qquad \text{(commuting conversion)}
\end{array}
$$

5. 1 elim

$$
\begin{array}{ll}
& \texttt{ret thunk}\ (\texttt{split}\ V\ \texttt{to}\ ().xyM) \\
\sqsupseteq\sqsubseteq & \texttt{split}\ V\ \texttt{to}\ ().\texttt{ret thunk split}\ ()\ \texttt{to}\ ().M \qquad (1\eta) \\
\sqsupseteq\sqsubseteq & \texttt{split}\ V\ \texttt{to}\ ().\texttt{ret thunk}\ M \qquad (1\beta) \\
\sqsupseteq\sqsubseteq & \texttt{split}\ V\ \texttt{to}\ ().z \leftarrow M; \texttt{ret thunk ret}\ z \\
& \qquad (M\ \text{thunkable}) \\
\sqsupseteq\sqsubseteq & z \leftarrow (\texttt{split}\ V\ \texttt{to}\ ().M); \texttt{ret thunk ret}\ z \\
& \qquad \text{(commuting conversion)}
\end{array}
$$

6. μ elim

$$\begin{aligned}
&\texttt{ret thunk } (\texttt{unroll } V \texttt{ to roll } x.M) \\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.\texttt{ret thunk unroll roll } x \texttt{ to roll } x.M && (\mu\eta) \\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.\texttt{ret thunk } M && (\mu\beta) \\
&\sqsupseteq\sqsubseteq \texttt{unroll } V \texttt{ to roll } x.y \leftarrow M; \texttt{ret thunk ret } y && (M \text{ thunkable}) \\
&\sqsupseteq\sqsubseteq y \leftarrow (\texttt{unroll } V \texttt{ to roll } x.M); \texttt{ret thunk ret } y && \text{(commuting conversion)}
\end{aligned}$$

□

Dually, we have that a stack out of a force is linear and that linears are closed under composition, so we can easily build up bigger linear morphisms from smaller ones.

Lemma 147 (Force to a stack is Linear)**.** *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$*, then* $\Gamma, x : U\underline{B} \vdash S[\texttt{force } x] : \underline{B}$ *is linear in* x*.*

Proof.

$$\begin{aligned}
S[\texttt{force thunk } (x \leftarrow \texttt{force } z; \texttt{force } x)] &\sqsupseteq\sqsubseteq S[(x \leftarrow \texttt{force } z; \texttt{force } x)] && (U\beta) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force } z; S[\texttt{force } x] && (\underline{F}\eta)
\end{aligned}$$

□

Lemma 148 (Linear Terms Compose)**.** *If* $\Gamma, x : U\underline{B} \vdash M : \underline{B}'$ *is linear in* x *and* $\Gamma, y : \underline{B}' \vdash N : \underline{B}''$ *is linear in* y*, then* $\Gamma, x : U\underline{B} \vdash N[\texttt{thunk } M/y] :$

Proof.

$$\begin{aligned}
&N[\texttt{thunk } M/y][\texttt{thunk } (x \leftarrow \texttt{force } z; \texttt{force } x)/x] \\
&= N[\texttt{thunk } (M[\texttt{thunk } (x \leftarrow \texttt{force } z; \texttt{force } x)])/y] \\
&\sqsupseteq\sqsubseteq N[\texttt{thunk } (x \leftarrow \texttt{force } z; M)/y] && (M \text{ linear}) \\
&\sqsupseteq\sqsubseteq N[\texttt{thunk } (x \leftarrow \texttt{force } z; \texttt{force thunk } M)/y] && (U\beta) \\
&\sqsupseteq\sqsubseteq N[\texttt{thunk } (x \leftarrow \texttt{force } z; y \leftarrow \texttt{ret thunk } M; \texttt{force } y)/y] && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq N[\texttt{thunk } (y \leftarrow (x \leftarrow \texttt{force } z; \texttt{ret thunk } M); \texttt{force } y)/y] && (\underline{F}\eta) \\
&\sqsupseteq\sqsubseteq N[\texttt{thunk } (y \leftarrow \texttt{force } w; \texttt{force } y)/y][\texttt{thunk } (x \leftarrow \texttt{force } z; \texttt{ret thunk } M)/w] && (U\beta) \\
&\sqsupseteq\sqsubseteq (y \leftarrow \texttt{force } w; N)[\texttt{thunk } (x \leftarrow \texttt{force } z; \texttt{ret thunk } M)/w] && (N \text{ linear}) \\
&\sqsupseteq\sqsubseteq (y \leftarrow (x \leftarrow \texttt{force } z; \texttt{ret thunk } M); N) && (U\beta) \\
&\sqsupseteq\sqsubseteq (x \leftarrow \texttt{force } z; y \leftarrow \texttt{ret thunk } M; N && (\underline{F}\eta) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force } z; N[\texttt{thunk } M/y]
\end{aligned}$$

□

Lemma 149 (Complex Stacks Simplify to Linear Terms)**.** *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$ *is a (possibly) complex stack, then* $\Gamma, z : U\underline{B} \vdash (S)^{\dagger} : \underline{C}$ *is linear in* z*.*

Proof. There are 4 classes of rules for complex stacks: those that are rules for simple stacks ($\bullet$, computation type elimination forms), introduction rules for negative computation types where the subterms are complex stacks, elimination of positive value types where the continuations are complex stacks and finally application to a complex value.

The rules for simple stacks are easy: they follow immediately from the fact that forcing to a stack is linear and that complex stacks compose. For the negative introduction forms, we have to show that binding commutes with introduction forms. For pattern matching forms, we just need commuting conversions. For function application, we use the lemma that binding a thunkable in a linear term is linear.

1. $\bullet$: This is just saying that $\texttt{force}\ z$ is linear, which we showed above.

2. $\to$ elim We need to show, assuming that $\Gamma, x : \underline{B} \vdash M : \underline{C}$ is linear in x and $\Gamma \vdash N : \underline{F}A$ is thunkable, that

   $$y \leftarrow N; M\, y$$

   is linear in x.

   $$\begin{aligned}
   & y \leftarrow N; (M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x])\, y \\
   \sqsupseteq\sqsubseteq\ & y \leftarrow N; (x \leftarrow \texttt{force}\ z; M)\, y && (M \text{ linear in } x) \\
   \sqsupseteq\sqsubseteq\ & y \leftarrow N; x \leftarrow \texttt{force}\ z; M\, y && (\underline{F}\eta) \\
   \sqsupseteq\sqsubseteq\ & x \leftarrow \texttt{force}\ z; y \leftarrow N; M\, y && (\text{thunkables are central})
   \end{aligned}$$

3. $\to$ intro

   $$\begin{aligned}
   & \lambda y : A.M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x] \\
   \sqsupseteq\sqsubseteq\ & \lambda y : A.x \leftarrow \texttt{force}\ z; M && (M \text{ is linear}) \\
   \sqsupseteq\sqsubseteq\ & \lambda y : A.x \leftarrow \texttt{force}\ z; (\lambda y : A.M)\, y && (\to \beta) \\
   \sqsupseteq\sqsubseteq\ & \lambda y : A.(x \leftarrow \texttt{force}\ z; (\lambda y : A.M))\, y && (\underline{F}\eta) \\
   \sqsupseteq\sqsubseteq\ & x \leftarrow \texttt{force}\ z; (\lambda y : A.M) && (\to \eta)
   \end{aligned}$$

4. $\top$ intro We need to show

   $$w \leftarrow \texttt{force}\ z; \{\} \sqsupseteq\sqsubseteq \{\}$$

   Which is immediate by $\top\eta$

5. & intro

$$
\begin{aligned}
&\{\pi \mapsto M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)]/x \\
&\mid \pi' \mapsto N[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto x \leftarrow \texttt{force}\ z; M && (M, N \text{ linear}) \\
&\quad \mid \pi' \mapsto x \leftarrow \texttt{force}\ z; N\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto x \leftarrow \texttt{force}\ z; \pi(M, N) && (\&\beta) \\
&\quad \mid \pi' \mapsto x \leftarrow \texttt{force}\ z; \pi'(M, N)\} \\
&\sqsupseteq\sqsubseteq \{\pi \mapsto \pi(x \leftarrow \texttt{force}\ z; (M, N)) && (\underline{F}\eta) \\
&\quad \mid \pi' \mapsto \pi'(x \leftarrow \texttt{force}\ z; (M, N))\} \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; (M, N) && (\&\eta)
\end{aligned}
$$

6. ν intro

$$
\begin{aligned}
&\texttt{roll}\ M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x] \\
&\sqsupseteq\sqsubseteq \texttt{roll}\ (x \leftarrow \texttt{force}\ z; M) && (M \text{ is linear}) \\
&\sqsupseteq\sqsubseteq \texttt{roll}\ (x \leftarrow \texttt{force}\ z; \texttt{unroll}\ \texttt{roll}\ M) && (\nu\beta) \\
&\sqsupseteq\sqsubseteq \texttt{roll}\ \texttt{unroll}\ (x \leftarrow \texttt{force}\ z; \texttt{roll}\ M) && (\underline{F}\eta) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; (\texttt{roll}\ M) && (\nu\eta)
\end{aligned}
$$

7. $\underline{F}$ elim: Assume $\Gamma, x : A \vdash M : \underline{F}A'$ and $\Gamma, y : A' \vdash N : \underline{B}$, then we need to show

   $$y \leftarrow M; N$$

   is linear in M.

   $$
   \begin{aligned}
   &y \leftarrow M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]; N \\
   &\sqsupseteq\sqsubseteq y \leftarrow (x \leftarrow \texttt{force}\ z; M); N && (M \text{ is linear}) \\
   &\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; y \leftarrow M; N && (\underline{F}\eta)
   \end{aligned}
   $$

8. 0 elim: We want to show $\Gamma, x : U\underline{B} \vdash \texttt{absurd}\ V : \underline{C}$ is linear in x, which means showing:

   $$\texttt{absurd}\ V \sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; \texttt{absurd}\ V$$

   which follows from 0η

9. $+$ elim: Assuming $\Gamma, x : U\underline{B}, y_1 : A_1 \vdash M_1 : \underline{C}$ and $\Gamma, x : U\underline{B}, y_2 : A_2 \vdash M_2 : \underline{C}$ are linear in x, and $\Gamma \vdash V : A_1 + A_2$, we need to show

   $$\texttt{case}\ V\{y_1.M_1 \mid y_2.M_2\}$$

is linear in x.

$$\begin{aligned}
&\texttt{case}\ V \\
&\{y_1.M_1[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x] \\
&\mid y_2.M_2[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]\} \\
&\sqsupseteq\sqsubseteq \texttt{case}\ V\{y_1.x \leftarrow \texttt{force}\ z; M_1 \mid y_2.x \leftarrow \texttt{force}\ z; M_2\} \\
&\qquad\qquad (M_1, M_2 \text{ linear}) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; \texttt{case}\ V\{y_1.M_1 \mid y_2.M_2\}
\end{aligned}$$

10. $\times$ elim: Assuming $\Gamma, x : U\underline{B}, y_1 : A_1, y_2 : A_2 \vdash M : \underline{B}$ is linear in x and $\Gamma \vdash V : A_1 \times A_2$, we need to show

$$\texttt{let}\ (y_1, y_2) = V; M$$

is linear in x.

$$\begin{aligned}
&\texttt{let}\ (y_1, y_2) = V; M[[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x]] \\
&\sqsupseteq\sqsubseteq \texttt{let}\ (y_1, y_2) = V; x \leftarrow \texttt{force}\ z; M \qquad (M \text{ linear}) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; \texttt{let}\ (y_1, y_2) = V; M \qquad (\text{comm. conv})
\end{aligned}$$

11. μ elim: Assuming $\Gamma, x : U\underline{B}, y : A[\mu X.A/X] \vdash M : \underline{C}$ is linear in x and $\Gamma \vdash V : \mu X.A$, we need to show

$$\texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M$$

is linear in x.

$$\begin{aligned}
&\texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M[\texttt{thunk}\ (x \leftarrow \texttt{force}\ z; \texttt{force}\ x)/x] \\
&\sqsupseteq\sqsubseteq \texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.x \leftarrow \texttt{force}\ z; M \qquad (M \text{ linear}) \\
&\sqsupseteq\sqsubseteq x \leftarrow \texttt{force}\ z; \texttt{unroll}\ V\ \texttt{to}\ \texttt{roll}\ y.M \\
&\qquad\qquad (\text{commuting conversion})
\end{aligned}$$

□

Composing this with the previous translation from GTT to CBPV* shows that *GTT value type upcasts are thunkable and computation type downcasts are linear*.

Since the translation takes values and stacks to terms, it cannot preserve substitution up to equality. Rather, we get the following, weaker notion that says that the translation of a syntactic substitution is equivalent to an effectful composition.

Lemma 150 (Compositionality of De-complexification). *1. If* $\Gamma, x : A \mid \Delta \vdash E : T$ *and* $\Gamma \vdash V : A$ *are complex terms, then*

$$(E[V/x])^\dagger \sqsupseteq\sqsubseteq x \leftarrow V^\dagger; E^\dagger$$

2. *If* $\Gamma \mid \bullet : \underline{B} \vdash S : \underline{C}$ *and* $\Gamma \mid \Delta \vdash M : \underline{B}$*, then*

$$(S[M])^\dagger \sqsupseteq\sqsubseteq S^\dagger[\mathsf{thunk}\ M^\dagger/z]$$

Proof. 1. First, note that every occurrence of a variable in $E^\dagger$ is of the form $\mathtt{ret}\ x$ for some variable x. This means we can define substitution of a *term* for a variable in a simplified term by defining $E^\dagger[N/\mathtt{ret}\ x]$ to replace every $\mathtt{ret}\ x : \underline{F}A$ with $N : \underline{F}A$. Then it is an easy observation that simplification is compositional on the nose with respect to this notion of substitution:

$$(E[V/x])^\dagger = E^\dagger[V^\dagger/\mathtt{ret}\ x]$$

Next by repeated invocation of $U\beta$,

$$E^\dagger[V^\dagger/\mathtt{ret}\ x] \sqsupseteq\sqsubseteq E^\dagger[\mathtt{force\ thunk}\ V^\dagger/\mathtt{ret}\ x]$$

Then we can lift the definition of the thunk to the top-level by $\underline{F}\beta$:

$$E^\dagger[\mathtt{force\ thunk}\ V^\dagger/\mathtt{ret}\ x] \sqsupseteq\sqsubseteq \mathtt{thunk}\ \leftarrow \mathtt{ret}\ ; V^\dagger w E^\dagger[\mathtt{force}\ w/\mathtt{ret}\ x]$$

Then because $V^\dagger$ is thunkable, we can bind it at the top-level and reduce an administrative redex away to get our desired result:

$$\begin{aligned}
&\mathtt{thunk}\ \leftarrow \mathtt{ret}\ ; V^\dagger w E^\dagger[\mathtt{force}\ w/\mathtt{ret}\ x] \\
&\sqsupseteq\sqsubseteq x \leftarrow V^\dagger; w \leftarrow \mathtt{ret\ thunk\ ret}\ x; E^\dagger[\mathtt{force}\ w/\mathtt{ret}\ x] && (V \text{ thunkable}) \\
&\sqsupseteq\sqsubseteq x \leftarrow V^\dagger; E^\dagger[\mathtt{force\ thunk\ ret}\ x/\mathtt{ret}\ x] && (\underline{F}\beta) \\
&\sqsupseteq\sqsubseteq x \leftarrow V^\dagger; E^\dagger[\mathtt{ret}\ x/\mathtt{ret}\ x] && (U\beta) \\
&\sqsupseteq\sqsubseteq x \leftarrow V^\dagger; E^\dagger
\end{aligned}$$

2. Note that every occurrence of z in $S^\dagger$ is of the form $\mathtt{force}\ z$. This means we can define substitution of a *term* $M : \underline{B}$ for $\mathtt{force}\ z$ in $S^\dagger$ by replacing $\mathtt{force}\ z$ with M. It is an easy observation that simplification is compositional on the nose with respect to this notion of substitution:

$$(S[M/\bullet])^\dagger = S^\dagger[M^\dagger/\mathtt{force}\ z]$$

Then by repeated $U\beta$, we can replace $M^\dagger$ with a forced thunk:

$$S^\dagger[M^\dagger/\mathtt{force}\ z] \sqsupseteq\sqsubseteq S^\dagger[\mathtt{force\ thunk}\ M^\dagger/\mathtt{force}\ z]$$

which since we are now substituting a force for a force is the same as substituting the thunk for the variable:

$$S^\dagger[\mathtt{force\ thunk}\ M^\dagger/\mathtt{force}\ z] \sqsupseteq\sqsubseteq S^\dagger[\mathtt{thunk}\ M^\dagger/z]$$

□

Finally we conclude with our desired theorem, that de-complexification preserves the precision relation.

Theorem 151 (De-complexification Preserves Precision)**.** *If* $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ *then* $\Gamma, \Delta^\dagger \vdash E^\dagger \sqsubseteq E'^\dagger : T^\dagger$

Proof. 1. Reflexivity is translated to reflexivity.

2. Transitivity is translated to transitivity.

3. Compatibility rules are translated to compatibility rules.

4. Substitution of a Value

$$\frac{\Gamma, x : A, \Delta^\dagger \vdash E^\dagger \sqsubseteq E'^\dagger : T^\dagger \qquad \Gamma \vdash V^\dagger \sqsubseteq V'^\dagger : \underline{F}A}{\Gamma, \Delta^\dagger \vdash E[V/x]^\dagger \sqsubseteq E'[V'/x]^\dagger : T^\dagger}$$

   By the compositionality lemma, it is sufficient to show:

$$x \leftarrow V^\dagger; E^\dagger \sqsubseteq x \leftarrow V'^\dagger; E'$$

   which follows by bind compatibility.

5. Plugging a term into a hole:

$$\frac{\Gamma, z : U\underline{C} \vdash S^\dagger \sqsubseteq S'^\dagger : \underline{B} \qquad \Gamma, \Delta^\dagger \vdash M^\dagger \sqsubseteq M'^\dagger : \underline{C}}{\Gamma, \Delta^\dagger \vdash S[M]^\dagger \sqsubseteq S'[M']^\dagger : \underline{B}}$$

   By compositionality, it is sufficient to show

$$S^\dagger[\texttt{thunk}\ M^\dagger/z] \sqsubseteq S'^\dagger[\texttt{thunk}\ M'^\dagger/z]$$

   which follows by thunk compatibility and the simple substitution rule.

6. Stack strictness We need to show for S a complex stack, that

$$(S[\mho])^\dagger \sqsupseteq \sqsubseteq \mho$$

   By stack compositionality we know

$$(S[\mho])^\dagger \sqsupseteq \sqsubseteq S^\dagger[\texttt{thunk}\ \mho/z]$$

$$\begin{aligned}
[\![S]\!][\texttt{thunk}\ \mho/z] &\sqsupseteq \sqsubseteq S^\dagger[\texttt{thunk}\ (y \leftarrow \mho; \mho)/z] && \text{(Stacks preserve } \mho\text{)} \\
&\sqsupseteq \sqsubseteq y \leftarrow \mho; S^\dagger[\texttt{thunk}\ \mho/z] && (S^\dagger \text{ is linear in } z) \\
&\sqsupseteq \sqsubseteq \mho && \text{(Stacks preserve } \mho\text{)}
\end{aligned}$$

7. 1β By compositionality it is sufficient to show

$$x \leftarrow \texttt{ret}\ (); \texttt{split}\ x\ \texttt{to}\ ().E^\dagger \sqsupseteq \sqsubseteq x \leftarrow \texttt{ret}\ (); E^\dagger$$

   which follows by $\underline{F}\beta, 1\beta$.

8. 1η We need to show for $\Gamma, x : 1 \mid \Delta \vdash E : T$

   $$E^\dagger \sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ x; \mathtt{split}\ x\ \mathtt{to}\ ().(E[()/x])^\dagger$$

   after a $\underline{F}\beta$, it is sufficient using 1η to prove:

   $$(E[()/x])^\dagger \sqsupseteq\sqsubseteq E^\dagger[()/x]$$

   which follows by compositionality and $\underline{F}\beta$:

   $$(E[()/x])^\dagger \sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ (); E^\dagger \sqsupseteq\sqsubseteq E^\dagger[()/x]$$

9. $\times\beta$ By compositionality it is sufficient to show

   $$\begin{aligned} &x \leftarrow (x_1 \leftarrow {V_1}^\dagger; x_2 \leftarrow {V_2}^\dagger; \mathtt{ret}\ (x_1, x_2)); \mathtt{let}\ (x_1, x_2) = x; E^\dagger \\ &\sqsupseteq\sqsubseteq x_1 \leftarrow {V_1}^\dagger; x_2 \leftarrow {V_2}^\dagger; E^\dagger \end{aligned}$$

   which follows by $\underline{F}\eta, \underline{F}\beta, \times\beta$.

10. $\times\eta$ We need to show for $\Gamma, x : A_1 \times A_2 \mid \Delta \vdash E : T$ that

    $$E^\dagger \sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ x; \mathtt{let}\ (x_1, x_2) = x; (E[(x_1, x_2)/x])^\dagger$$

    by $\underline{F}\beta, \times\eta$ it is sufficient to show

    $$E[(x_1, x_2)/x]^\dagger \sqsupseteq\sqsubseteq E^\dagger[(x_1, x_2)/x]$$

    Which follows by compositionality:

    $$\begin{aligned} &E[(x_1, x_2)/x]^\dagger \\ &\sqsupseteq\sqsubseteq x_1 \leftarrow x_1; x_2 \leftarrow x_2; x \leftarrow \mathtt{ret}\ (x_1, x_2); E^\dagger && \text{(compositionality)} \\ &\sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ (x_1, x_2); E^\dagger && (\underline{F}\beta) \\ &\sqsupseteq\sqsubseteq E^\dagger[(x_1, x_2)/x] \end{aligned}$$

11. 0η We need to show for any $\Gamma, x : 0 \mid \Delta \vdash E : T$ that

    $$E^\dagger \sqsupseteq\sqsubseteq x \leftarrow \mathtt{ret}\ x; \mathtt{absurd}\ x$$

    which follows by 0η

12. $+\beta$ Without loss of generality, we do the `inl` case By compositionality it is sufficient to show

    $$x \leftarrow (x \leftarrow V^\dagger; \mathtt{inl}\ x); \mathtt{case}\ x\{x_1.E_1^\dagger \mid x_2.E_2^\dagger\} \sqsupseteq\sqsubseteq E_1[V/x_1]^\dagger$$

    which holds by $\underline{F}\eta, \underline{F}\beta, +\beta$

13. $+\eta$ We need to show for any $\Gamma, x : A_1 + A_2 \mid \Delta \vdash E : T$ that

$$E^\dagger \sqsupseteq\sqsubseteq x \leftarrow \texttt{ret}\ x; \texttt{case}\ x\{x_1.(E[\texttt{inl}\ x_1/x])^\dagger \mid x_2.(E[\texttt{inl}\ x_2/x])^\dagger\}$$

$$\begin{array}{ll}
& E^\dagger \\
\sqsupseteq\sqsubseteq & \texttt{case}\ x\{x_1.E^\dagger[\texttt{inl}\ x_1/x] \mid x_2.E^\dagger[\texttt{inl}\ x_2/x]\} \qquad (+\eta) \\
\sqsupseteq\sqsubseteq & \texttt{case}\ x\{x_1.x \leftarrow \texttt{ret}\ \texttt{inl}\ x_1; E^\dagger \mid x_2.x \leftarrow \texttt{ret}\ \texttt{inl}\ x_2; E^\dagger\} \\
& \qquad\qquad (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq & \texttt{case}\ x\{x_1.E[\texttt{inl}\ x_1]/x^\dagger \mid x_2.E[\texttt{inl}\ x_2]/x^\dagger\} \\
& \qquad\qquad \text{(compositionality)} \\
\sqsupseteq\sqsubseteq & x \leftarrow \texttt{ret}\ x; \texttt{case}\ x\{x_1.E[\texttt{inl}\ x_1]/x^\dagger \mid x_2.E[\texttt{inl}\ x_2]/x^\dagger\} \\
& \qquad\qquad (\underline{F}\beta)
\end{array}$$

14. $\mu\beta$ By compositionality it is sufficient to show

$$\begin{array}{l}
x \leftarrow (y \leftarrow V^\dagger; \texttt{ret}\ \texttt{roll}\ y); \texttt{unroll}\ x\ \texttt{to}\ \texttt{roll}\ y.E \\
\sqsupseteq\sqsubseteq y \leftarrow V^\dagger; E^\dagger
\end{array}$$

which follows by $\underline{F}\eta, \underline{F}\beta, \mu\beta$.

15. $\mu\eta$ We need to show for $\Gamma, x : \mu X.A \mid \Delta \vdash E : T$ that

$$E^\dagger \sqsupseteq\sqsubseteq x \leftarrow \texttt{ret}\ x; \texttt{unroll}\ x\ \texttt{to}\ \texttt{roll}\ y.(E[\texttt{roll}\ y/x])^\dagger$$

by $\underline{F}\beta, \times\eta$ it is sufficient to show

$$E[\texttt{roll}\ y/x]^\dagger \sqsupseteq\sqsubseteq E^\dagger[\texttt{roll}\ y/x]$$

Which follows by compositionality:

$$\begin{array}{ll}
& E[\texttt{roll}\ y/x]^\dagger \\
\sqsupseteq\sqsubseteq & y \leftarrow \texttt{ret}\ y; x \leftarrow \texttt{ret}\ \texttt{roll}\ y; E^\dagger \qquad \text{(compositionality)} \\
\sqsupseteq\sqsubseteq & x \leftarrow \texttt{ret}\ \texttt{roll}\ y; E^\dagger \qquad (\underline{F}\beta) \\
\sqsupseteq\sqsubseteq & E^\dagger[\texttt{roll}\ y/x] \qquad (\underline{F}\beta)
\end{array}$$

16. $U\beta$ We need to show

$$x \leftarrow \texttt{ret}\ M^\dagger; \texttt{force}\ x \sqsupseteq\sqsubseteq M^\dagger$$

which follows by $\underline{F}\beta, U\beta$

17. $U\eta$ We need to show for any $\Gamma \vdash V : U\underline{B}$ that

$$V^\dagger \sqsupseteq\sqsubseteq \texttt{ret}\ \texttt{thunk}\ (x \leftarrow V^\dagger; \texttt{force}\ x)$$

By compositionality it is sufficient to show

$$V^\dagger \sqsupseteq\sqsubseteq x \leftarrow V^\dagger; \texttt{ret}\ \texttt{thunk}\ (x \leftarrow \texttt{ret}\ x; \texttt{force}\ x)$$

which follows by $U\eta$ and some simple reductions:

$$
\begin{array}{ll}
x \leftarrow V^\dagger; \texttt{ret thunk } (x \leftarrow \texttt{ret } x; \texttt{force } x) & \\
\qquad \sqsupseteq\sqsubseteq x \leftarrow V^\dagger; \texttt{ret thunk force } x & \\
\qquad\qquad (\underline{F}\beta) & \\
\qquad \sqsupseteq\sqsubseteq x \leftarrow V^\dagger; \texttt{ret } x & \\
\qquad\qquad (U\eta) & \\
\qquad \sqsupseteq\sqsubseteq V^\dagger & (\underline{F}\eta)
\end{array}
$$

18. $\to\beta$ By compositionality it is sufficient to show

    $$x \leftarrow V^\dagger; (\lambda x : A.M^\dagger)\, x \sqsupseteq\sqsubseteq x \leftarrow V^\dagger; M^\dagger$$

    which follows by $\to\beta$

19. $\to\eta$ We need to show

    $$z : U(A \to \underline{B}) \vdash \texttt{force } z \sqsupseteq\sqsubseteq \lambda x : A.x \leftarrow \texttt{ret } x; (\texttt{force } z)\, x$$

    which follows by $\underline{F}\beta, \to\eta$

20. $\top\eta$ We need to show

    $$z : U\top \vdash \texttt{force } z \sqsupseteq\sqsubseteq \{\}$$

    which is exactly $\top\eta$.

21. $\&\beta$ Immediate by simple $\&\beta$.

22. $\&\eta$ We need to show

    $$z : U(\underline{B}_1 \,\&\, \underline{B}_2) \vdash \texttt{force } z \sqsupseteq\sqsubseteq (\pi\texttt{force } z, \pi'\texttt{force } z)$$

    which is exactly $\&\eta$

23. $\nu\beta$ Immediate by simple $\nu\beta$

24. $\nu\eta$ We need to show

    $$z : U(\nu\underline{Y}.\underline{B}) \vdash \texttt{force } z \sqsupseteq\sqsubseteq \texttt{roll unroll } z$$

    which is exactly $\nu\eta$

25. $\underline{F}\beta$ We need to show

    $$x \leftarrow V^\dagger; M^\dagger \sqsupseteq\sqsubseteq M[V/x]^\dagger$$

    which is exactly the compositionality lemma.

26. $\underline{F}\eta$ We need to show

    $$z : U(\underline{F}A)\texttt{force } z \vdash x \leftarrow \texttt{force } z; x \leftarrow \texttt{ret } x; \texttt{ret } x$$

    which follows by $\underline{F}\beta, \underline{F}\eta$

□

As a corollary, we also get the following conservativity result that says that precision in CBPV with complex values and stacks coincides with CBPV without them. This shows that complex values and stacks can be viewed as simply a convenient way to manipulate thunkable and linear terms and the calculus is not fundamentally different from CBPV.

Corollary 152 (Complex CBPV is Conservative over CBPV)**.** *If M, M' are terms in CBPV and $M \sqsubseteq M'$ is provable in CBPV* then $M \sqsubseteq M'$ is provable in CBPV.*

Proof. Because de-complexification preserves precision, $M^\dagger \sqsubseteq M'^\dagger$ in simple CBPV. Then it follows because de-complexification is equivalent to identity (in CBPV):

$$M \sqsupseteq\sqsubseteq M^\dagger \sqsubseteq M'^\dagger \sqsupseteq\sqsubseteq M'$$

□

7.4 OPERATIONAL MODEL OF GTT

In this section, we establish a model of our CBPV inequational theory using a notion of observational approximation based on the CBPV operational semantics. By composition with the axiomatic graduality theorem, this establishes the *operational graduality* theorem, i.e., a theorem analogous to the *dynamic gradual guarantee* [75].

7.4.1 *Call-by-Push-Value Operational Semantics*

We use a small-step operational semantics for CBPV in Figure 7.11.

This is morally the same as in Levy [45], but we present stacks in a manner similar to Hieb-Felleisen style evaluation contexts(rather than as an explicit stack machine with stack frames). We also make the step relation count unrollings of a recursive or corecursive type, for the step-indexed logical relation later. The operational semantics is only defined for terms of type $\cdot \vdash M : \underline{F}(1+1)$, which we take as the type of whole programs.

We can then observe the following standard operational properties. (We write $M \mapsto N$ with no index when the index is irrelevant.)

Lemma 153 (Reduction is Deterministic)**.** *If $M \mapsto M_1$ and $M \mapsto M_2$, then $M_1 = M_2$.*

Lemma 154 (Subject Reduction)**.** *If $\cdot \vdash M : \underline{F}A$ and $M \mapsto M'$ then $\cdot \vdash M' : \underline{F}A$.*

Lemma 155 (Progress)**.** *If $\cdot \vdash M : \underline{F}A$ then one of the following holds:*

$$M = \mathcal{U} \qquad M = \mathtt{ret}\ V \mathtt{with} V : A \qquad \exists M'.\ M \mapsto M'$$

$$
\begin{array}{rcl}
S[\mho] & \mapsto^0 & \mho \\
S[\texttt{case inl } V\{x_1.M_1 \mid x_2.M_2\}] & \mapsto^0 & S[M_1[V/x_1]] \\
S[\texttt{case inr } V\{x_1.M_1 \mid x_2.M_2\}] & \mapsto^0 & S[M_2[V/x_2]] \\
S[\texttt{let } (x_1, x_2) = (V_1, V_2); M] & \mapsto^0 & S[M[V_1/x_1, V_2/x_2]] \\
S[\texttt{unroll roll}_A\ V \texttt{ to roll } x.M] & \mapsto^1 & S[M[V/x]] \\
S[\texttt{force thunk } M] & \mapsto^0 & S[M] \\
S[\texttt{let } x = V; M] & \mapsto^0 & S[M[V/x]] \\
S[x \leftarrow \texttt{ret } V; M] & \mapsto^0 & S[M[V/x]] \\
S[(\lambda x : A.M)\, V] & \mapsto^0 & S[M[V/x]] \\
S[\pi(M, M')] & \mapsto^0 & S[M] \\
S[\pi'(M, M')] & \mapsto^0 & S[M'] \\
S[\texttt{unroll roll}_{\underline{B}}\ M] & \mapsto^1 & S[M]
\end{array}
$$

$$
\frac{}{M \mapsto^0 M} \qquad \frac{M_1 \mapsto^i M_2 \qquad M_2 \mapsto^j M_3}{M_1 \mapsto^{i+j} M_3}
$$

Figure 7.11: CBPV Operational Semantics

The standard progress-and-preservation properties allow us to define the "final result" of a computation as follows:

Corollary 156 (Possible Results of Computation)**.** *For any* $\cdot \vdash M : \underline{F}2$*, one of the following is true:*

$$M \Uparrow \qquad M \Downarrow \mho \qquad M \Downarrow \texttt{ret true} \qquad M \Downarrow \texttt{ret false}$$

Proof. We define $M \Uparrow$ to hold when if $M \mapsto^i N$ then there exists N' with $N \mapsto N'$. For the terminating results, we define $M \Downarrow R$ to hold if there exists some i with $M \mapsto^i R$. Then we prove the result by coinduction on execution traces. If $M \in \{\mho, \texttt{ret true}, \texttt{ret false}\}$ then we are done, otherwise by progress, $M \mapsto M'$, so we need only observe that each of the cases above is preserved by $\mapsto$. □

Definition 157 (Results)**.** The possible results of a computation are $\Omega, \mho,$ `ret true` and `ret false`. We denote a result by R, and define a function result which takes a program $\cdot \vdash M : \underline{F}2$, and returns its end-behavior, i.e., $\text{result}(M) = \Omega$ if $M \Uparrow$ and otherwise $M \Downarrow \text{result}(M)$.

7.4.2 *Observational Equivalence and Approximation*

Next, we define observational equivalence and approximation in CBPV. The (standard) definition of observational equivalence is that we consider two terms (or values) to be equivalent when replacing one

$$
\begin{array}{rcl}
C_V & ::= & [\cdot] \mid \mathtt{roll}_{\mu X.A}\ C_V \mid \mathtt{inl}\ C_V \mid \mathtt{inr}\ C_V \mid (C_V, V) \mid (V, C_V) \mid \mathtt{thunk}\ C_M \\
\\
C_M & ::= & [\cdot] \mid \mathtt{let}\ x = C_V; M \mid \mathtt{let}\ x = V; C_M \mid \mathtt{unroll}\ C_V\ \mathtt{to}\ \mathtt{roll}\ x.M \\
& \mid & \mathtt{unroll}\ V\ \mathtt{to}\ \mathtt{roll}\ x.C_M \mid \mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ C_M \mid \mathtt{unroll}\ C_M \mid \mathtt{abort}\ C_V \\
& \mid & \mathtt{case}\ C_V\{x_1.M_1 \mid x_2.M_2\} \mid \mathtt{case}\ V\{x_1.C_M \mid x_2.M_2\} \mid \mathtt{case}\ V\{x_1.M_1 \mid x_2.C_M\} \\
& \mid & \mathtt{split}\ C_V\ \mathtt{to}\ ().M \mid \mathtt{split}\ V\ \mathtt{to}\ ().C_M \mid \mathtt{let}\ (x,y) = C_V; M \\
& \mid & \mathtt{let}\ (x,y) = V; C_M \mid \mathtt{force}\ C_V \mid \mathtt{ret}\ C_V \mid x \leftarrow C_M; N \\
& \mid & x \leftarrow M; C_M \mid \lambda x : A.C_M \mid C_M\, V \mid M\, C_V \\
& \mid & (C_M, M_2) \mid (M_1, C_M) \mid \pi C_M \mid \pi' C_M \\
C_S & ::= & \pi C_S \mid \pi' C_S \mid S\, C_V \mid C_S\, V \mid x \leftarrow C_S; M \mid x \leftarrow S; C_M
\end{array}
$$

Figure 7.12: CBPV Contexts

with the other in any program text produces the same overall resulting computation. Define a context C to be a term/value/stack with a single $[\cdot]$ as some subterm/value/stack, and define a typing $C : (\Gamma \vdash \underline{B}) \Rightarrow (\Gamma' \vdash \underline{B}')$ to hold when for any $\Gamma \vdash M : \underline{B}$, $\Gamma' \vdash C[M] : \underline{B}'$ (and similarly for values/stacks). Using contexts, we can lift any relation on *results* to relations on open terms, values and stacks.

Definition 158 (Contextual Lifting)**.** Given any relation $\sim \subseteq \mathrm{Result}^2$, we can define its *observational lift* $\sim^{\mathrm{ctx}}$ to be the typed relation defined by

$$\Gamma \mid \Delta \vDash E \sim^{\mathrm{ctx}} E' \in T = \forall C : (\Gamma \mid \Delta \vdash T) \Rightarrow (\cdot \vdash \underline{F}2).\ \mathrm{result}(C[E]) \sim \mathrm{result}(C[E'])$$

The contextual lifting $\sim^{\mathrm{ctx}}$ inherits much structure of the original relation $\sim$ as the following lemma shows. This justifies calling $\sim^{\mathrm{ctx}}$ a contextual preorder when $\sim$ is a preorder (reflexive and transitive) and similarly a contextual equivalence when $\sim$ is an equivalence (preorder and symmetric).

Definition 159 (Contextual Preorder, Equivalence)**.** If $\sim$ is reflexive, symmetric or transitive, then for each typing, $\sim^{\mathrm{ctx}}$ is reflexive, symmetric or transitive as well, respectively.

In the remainder of the paper we work only with relations that are at least preorders so we write $\trianglelefteq$ rather than $\sim$.

The most famous use of lifting is for observational equivalence, which is the lifting of equality of results ($=^{\mathrm{ctx}}$), and we will show that $\sqsupseteq\sqsubseteq$ proofs in GTT imply observational equivalences. However, as shown in New and Ahmed [56], the graduality property is defined in terms of an observational *approximation* relation $\sqsubseteq$ that places $\mho$ as the least element, and every other element as a maximal element. Note that this is *not* the standard notion of observational approximation, which

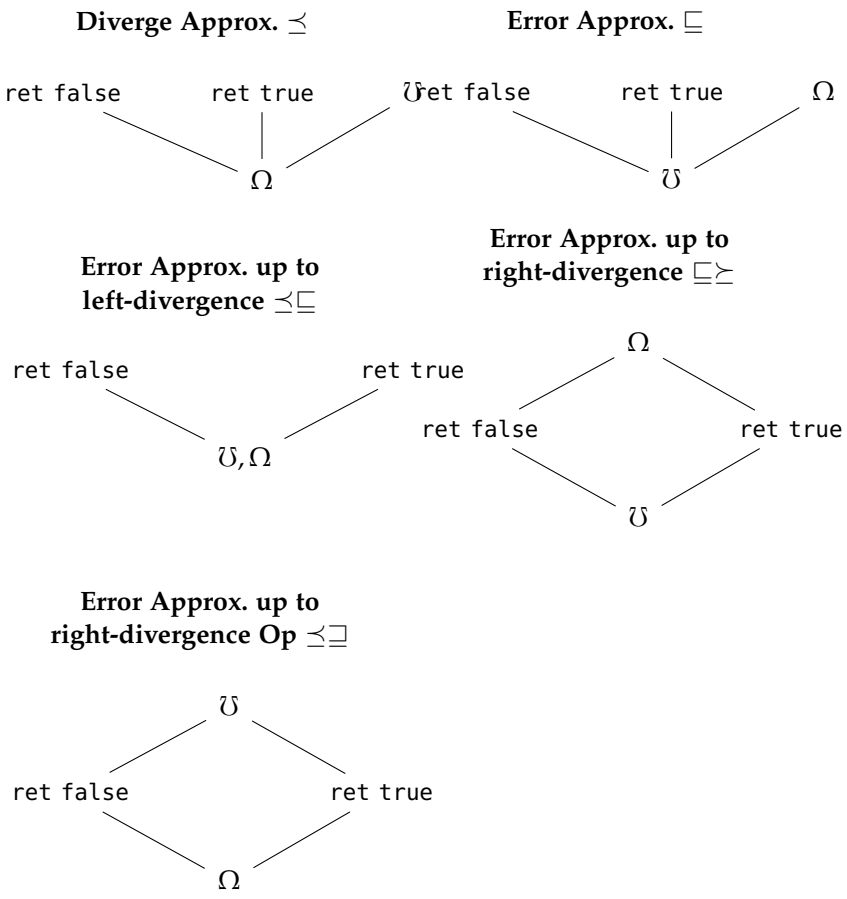

Figure 7.13: Result Orderings

we write $\preceq$, which makes Ω a least element and every other element a maximal element. To distinguish these, we call $\sqsubseteq$ *error* approximation and $\preceq$ *divergence* approximation. We present these graphically (with two more) in Figure 7.13.

The goal of this section is to prove that a symmetric equality $E \sqsupseteq\sqsubseteq E'$ in CBPV (i.e. $E \sqsubseteq E'$ and $E' \sqsubseteq E$) implies contextual equivalence $E =^{\text{ctx}} E'$ and that inequality in CBPV $E \sqsubseteq E'$ implies error approximation $E \sqsubseteq^{\text{ctx}} E'$, proving graduality of the operational model:

$$\frac{\Gamma \mid \Delta \vdash E \sqsupseteq\sqsubseteq E' : T}{\Gamma \mid \Delta \vDash E =^{\text{ctx}} E' \in T} \qquad \frac{\Gamma \mid \Delta \vdash E \sqsubseteq E' : T}{\Gamma \mid \Delta \vDash E \sqsubseteq^{\text{ctx}} E' \in T}$$

Because we have non-well-founded μ/ν types, we use a *step-indexed logical relation* to prove properties about the contextual lifting of certain preorders $\trianglelefteq$ on results. In step-indexing, the *infinitary* relation given by $\trianglelefteq^{\text{ctx}}$ is related to the set of all of its *finitary approximations* $\trianglelefteq^{i}$, which "time out" after observing i steps of evaluation and declare that the terms *are* related. This means that the original relation is only recoverable from the finite approximations if Ω is always related to

another element: if the relation is a preorder, we require that Ω is a *least* element.

We call such a preorder a *divergence preorder*.

Definition 160 (Divergence Preorder)**.** A preorder on results $\trianglelefteq$ is a divergence preorder if $\Omega \trianglelefteq R$ for all results R.

But this presents a problem, because *neither* of our intended relations ($=$ and $\sqsubseteq$) is a divergence preorder; rather both have Ω as a *maximal* element. However, there is a standard "trick" for subverting this obstacle in the case of contextual equivalence [2]: we notice that we can define equivalence as the symmetrization of divergence approximation, i.e., $M =^{\text{ctx}} N$ if and only if $M \preceq^{\text{ctx}} N$ and $N \preceq^{\text{ctx}} M$, and since $\preceq$ has Ω as a least element, we can use a step-indexed relation to prove it. As shown in New and Ahmed [56], a similar trick works for error approximation, but since $\sqsubseteq$ is *not* an equivalence relation, we decompose it rather into two *different* orderings: error approximation up to divergence on the left $\preceq\sqsubseteq$ and error approximation up to divergence on the right $\sqsubseteq\succeq$, also shown in Figure 7.13. Note that $\preceq\sqsubseteq$ is a preorder, but not a poset because $\mho, \Omega$ are order-equivalent but not equal. Then clearly $\preceq\sqsubseteq$ is a divergence preorder and the *opposite* of $\sqsubseteq\succeq$, written $\preceq\sqsupseteq$ is a divergence preorder.

Then we can completely reduce the problem of proving $=^{\text{ctx}}$ and $\sqsubseteq^{\text{ctx}}$ results to proving results about divergence preorders by the following observations.

Lemma 161 (Decomposing Result Preorders)**.** *Let R, S be results.*

1. *$R = S$ if and only if $R \sqsubseteq S$ and $S \sqsubseteq R$.*
2. *$R = S$ if and only if $R \preceq S$ and $S \preceq R$.*
3. *$R \preceq\sqsubseteq S$ iff $R \sqsubseteq S$ or $R \preceq S$.*
4. *$R \sqsubseteq\succeq S$ iff $R \sqsubseteq S$ or $R \succeq S$.*

To prove Lemma 161, we develop a few lemmas about the interaction between contextual lifting and operations on relations.

In the following, we write $\sim^\circ$ for the opposite of a relation ($x \sim^\circ y$ iff $y \sim x$), $\Rightarrow$ for containment/implication ($\sim \Rightarrow \sim'$ iff $x \sim y$ implies $x \sim' y$), $\Leftrightarrow$ for bicontainment/equality, $\vee$ for union ($x(\sim \vee \sim')y$ iff $x \sim y$ or $x \sim' y$), and $\wedge$ for intersection ($x(\sim \wedge \sim')y$ iff $x \sim y$ and $x \sim' y$).

Lemma 162 (Contextual Lift commutes with Conjunction)**.**

$$(\sim_1 \wedge \sim_2)^{ctx} \Leftrightarrow {\sim_1}^{ctx} \wedge {\sim_2}^{ctx}$$

Lemma 163 (Contextual Lift commutes with Dualization)**.**

$$\sim^{\circ ctx} \Leftrightarrow \sim^{ctx\circ}$$

As a corollary, the decomposition of contextual equivalence into diverge approximation in Ahmed [2] and the decomposition of precision in New and Ahmed [56] are really the same trick:

Lemma 164 (Contextual Decomposition Lemma)**.** *Let $\sim$ be a reflexive relation $(=\Rightarrow\sim)$, and $\leqslant$ be a reflexive, antisymmetric relation $(= \Rightarrow \leqslant$ and $(\leqslant \wedge \leqslant^\circ) \Leftrightarrow =)$. Then*

$$\sim^{ctx} \Leftrightarrow (\sim \vee \leqslant)^{ctx} \wedge ((\sim^\circ \vee \leqslant)^{ctx})^\circ$$

Proof. Note that despite the notation, $\leqslant$ need not be assumed to be transitive. Reflexive relations form a lattice with $\wedge$ and $\vee$ with $=$ as $\bot$ and the total relation as $\top$ (e.g. $(= \vee \sim) \Leftrightarrow \sim$ because $\sim$ is reflexive, and $(= \wedge \sim) \Leftrightarrow =$). So we have

$$\sim \Leftrightarrow (\sim \vee \leqslant) \wedge (\sim \vee \leqslant^\circ)$$

because expanding the right-hand side gives

$$(\sim \wedge \sim) \vee (\leqslant \wedge \sim) \vee (\sim \wedge \leqslant^\circ) \vee (\leqslant \wedge \leqslant^\circ)$$

By antisymmetry, $(\leqslant \wedge \leqslant^\circ)$ is $=$, which is the unit of $\vee$, so it cancels. By idempotence, $(\sim \wedge \sim)$ is $\sim$. Then by absorption, the whole thing is $\sim$.

Opposite is *not* de Morgan: $(P \vee Q)^\circ = P^\circ \vee Q^\circ$, and similarly for $\wedge$. But it is involutive: $(P^\circ)^\circ \Leftrightarrow P$.

So using Lemmas 162, 163 we can calculate as follows:

$$\begin{aligned}
\sim^{\text{ctx}} &\Leftrightarrow ((\sim \vee \leqslant) \wedge (\sim \vee \leqslant^\circ))^{\text{ctx}} \\
&\Leftrightarrow (\sim \vee \leqslant)^{\text{ctx}} \wedge (\sim \vee \leqslant^\circ)^{\text{ctx}} \\
&\Leftrightarrow (\sim \vee \leqslant)^{\text{ctx}} \wedge ((\sim \vee \leqslant^\circ)^\circ)^{\circ\text{ctx}} \\
&\Leftrightarrow (\sim \vee \leqslant)^{\text{ctx}} \wedge ((\sim^\circ \vee (\leqslant^\circ)^\circ)^\circ)^{\text{ctx}} \\
&\Leftrightarrow (\sim \vee \leqslant)^{\text{ctx}} \wedge (\sim^\circ \vee \leqslant)^{\circ\text{ctx}} \\
&\Leftrightarrow (\sim \vee \leqslant)^{\text{ctx}} \wedge (\sim^\circ \vee \leqslant)^{\text{ctx}\circ}
\end{aligned}$$

□

7.4.3 *CBPV Step Indexed Logical Relation*

Next, we turn to the problem of proving results about $E \trianglelefteq^{\text{ctx}} E'$ where $\trianglelefteq$ is a divergence preorder. Dealing directly with a contextual preorder is practically impossible, so instead we develop an alternative formulation as a logical relation that is much easier to use. Fortunately, we can apply standard logical relations techniques to provide an alternate definition *inductively* on types. However, since we have non-well-founded type definitions using μ and ν, our logical relation will also be defined inductively on a *step index* that times out when we've exhausted our step budget. To bridge the gap between the indexed

logical relation and the divergence preorder we care about, we define the "finitization" of a divergence preorder to be a relation between *programs* and *results*: the idea is that a program approximates a result R at index i if it reduces to R in less than i steps or it reduces at least i times.

Definition 165 (Finitized Preorder)**.** Given a divergence preorder $\trianglelefteq$, we define the *finitization* of $\trianglelefteq$ to be, for each natural number i, a relation between programs and results

$$\trianglelefteq^i \subseteq \{M \mid \cdot \vdash M : \underline{F}2\} \times \text{Results}$$

defined by

$$M \trianglelefteq^i R = (\exists M'.\, M \mapsto^i M') \vee (\exists (j < i).\exists R_M.\, M \mapsto^j R_M \wedge R_M \trianglelefteq R)$$

Note that in this definition, unlike in the definition of divergence, we only count non-well-founded steps. This makes it slightly harder to establish the intended equivalence $M \trianglelefteq^\omega R$ if and only if $\text{result}(M) \trianglelefteq R$, but makes the logical relation theorem stronger: it proves that diverging terms must use recursive types of some sort and so any term that does not use them terminates. This issue would be alleviated if we had proved type safety by a logical relation rather than by progress and preservation.

However, the following properties of the indexed relation can easily be established. First, a kind of "transitivity" of the indexed relation with respect to the original preorder, which is key to proving transitivity of the logical relation.

Lemma 166 (Indexed Relation is a Module of the Preorder)**.** *If $M \trianglelefteq^i R$ and $R \trianglelefteq R'$ then $M \trianglelefteq^i R'$*

Proof. If $M \mapsto^i M'$ then there's nothing to show, otherwise $M \mapsto^{j<i} \text{result}(M)$ so it follows by transitivity of the preorder: $\text{result}(M) \trianglelefteq R \trianglelefteq R'$. □

Then we establish a few basic properties of the finitized preorder.

Lemma 167 (Downward Closure of Finitized Preorder)**.** *If $M \trianglelefteq^i R$ and $j \leq i$ then $M \trianglelefteq^j R$.*

Proof.

1. If $M \mapsto^i M_i$ then $M \mapsto^j M_j$ and otherwise
2. If $M \mapsto^{j \leq ki} \text{result}(M)$ then $M \mapsto^j M_j$
3. if $M \mapsto^{k<j\leq i} \text{result}(M)$ then $\text{result}(M) \trianglelefteq R$.

□

Lemma 168 (Triviality at 0)**.** *For any $\cdot \vdash M : \underline{F}2$, $M \trianglelefteq^0 R$*

$$\preceq^{\log}_{A,i} \subseteq \{\cdot \vdash V : A\}^2 \qquad \preceq^{\log}_{\underline{B},i} \subseteq \{\cdot \mid \underline{B} \vdash S : \underline{F}(1+1)\}^2$$

$$\begin{array}{rcl}
\cdot \preceq^{\log}_{\cdot,i} \cdot & = & \top \\
\gamma_1, V_1/x \preceq^{\log}_{\Gamma,x:A,i} \gamma_2, V_2/x & = & \gamma_1 \preceq^{\log}_{\Gamma,i} \gamma_2 \wedge V_1 \preceq^{\log}_{A,i} V_2 \\
\\
V_1 \preceq^{\log}_{0,i} V_2 & = & \bot \\
\mathtt{inl}\ V_1 \preceq^{\log}_{A+A',i} \mathtt{inl}\ V_2 & = & V_1 \preceq^{\log}_{A,i} V_2 \\
\mathtt{inr}\ V_1 \preceq^{\log}_{A+A',i} \mathtt{inr}\ V_2 & = & V_1 \preceq^{\log}_{A',i} V_2 \\
() \preceq^{\log}_{1,i} () & = & \top \\
(V_1, V_1') \preceq^{\log}_{A\times A',i} (V_2, V_2') & = & V_1 \preceq^{\log}_{A,i} V_2 \wedge V_1' \preceq^{\log}_{A',i} V_2' \\
\mathtt{roll}_{\mu X.A}\ V_1 \preceq^{\log}_{\mu X.A,i} \mathtt{roll}_{\mu X.A}\ V_2 & = & i = 0 \vee V_1 \preceq^{\log}_{A[\mu X.A/X],i-1} V_2 \\
V_1 \preceq^{\log}_{U\underline{B},i} V_2 & = & \forall j \leq i, S_1 \preceq^{\log}_{\underline{B},j} S_2. \\
& & \quad S_1[\mathtt{force}\ V_1] \preceq^{j} \mathrm{result}(S_2[\mathtt{force}\ V_2]) \\
\\
S_1[\bullet V_1] \preceq^{\log}_{A\to\underline{B},i} S_1[\bullet V_2] & = & V_1 \preceq^{\log}_{A,i} V_2 \wedge S_1 \preceq^{\log}_{\underline{B},i} S_2 \\
S_1[\pi_1 \bullet] \preceq^{\log}_{\underline{B}\&\underline{B}',i} S_2[\pi_1 \bullet] & = & S_1 \preceq^{\log}_{\underline{B},i} S_2 \\
S_1[\pi_2 \bullet] \preceq^{\log}_{\underline{B}\&\underline{B}',i} S_2[\pi_2 \bullet] & = & S_1 \preceq^{\log}_{\underline{B}',i} S_2 \\
S_1 \preceq^{\log}_{\top,i} S_2 & = & \bot \\
S_1[\mathtt{unroll}\ \bullet] \preceq^{\log}_{\nu\underline{Y}.\underline{B},i} S_2[\mathtt{unroll}\ \bullet] & = & i = 0 \vee S_1 \preceq^{\log}_{\underline{B}[\nu\underline{Y}.\underline{B}/\underline{Y}],i-1} S_2 \\
S_1 \preceq^{\log}_{\underline{F}A,i} S_2 & = & \forall j \leq i, V_1 \preceq^{\log}_{A,j} V_2. \\
& & \quad S_1[\mathtt{ret}\ V_1] \preceq^{j} \mathrm{result}(S_2[\mathtt{ret}\ V_2])
\end{array}$$

Figure 7.14: Logical Relation from a Preorder $\preceq$

Proof. Because $M \mapsto^0 M$ □

Lemma 169 (Result (Anti-)reduction)**.** *If $M \mapsto^i N$ then $result(M) = result(N)$.*

Lemma 170 (Anti-reduction)**.** *If $M \preceq^i R$ and $N \mapsto^j M$, then $N \preceq^{i+j} R$*

Proof.

1. If $M \mapsto^i M'$ then $N \mapsto^{i+j} M'$
2. If $M \mapsto^{k<i} \mathrm{result}(M)$ then $N \mapsto^{k+j} \mathrm{result}(M)$ and $\mathrm{result}(M) = \mathrm{result}(N)$ and $k + j < i + j$.

□

Next, we define the (closed) *logical* preorder (for closed values/stacks) by induction on types and the index i in Figure 7.14. Specifically, for every i and value type A we define a relation $\preceq^{\log}_{A,i}$ between closed values of type A because these are the only ones that will be pattern-matched against at runtime. The relation is defined in a type-directed fashion, the intuition being that we relate two positive values when

they are built up in the same way: i.e., they have the same introduction form and their subterms are related. For μ, this definition would not be well-founded, so we decrement the step index, giving up and relating the terms if $i = 0$. Finally U is the only negative value type, and so it is treated differently. A thunk $V : U\underline{B}$ cannot be inspected by pattern matching, rather the only way to interact with it is to force its evaluation. By the definition of the operational semantics, this only ever occurs in the step $S[\texttt{force}\ V]$, so (ignoring indices for a moment), we should define $V_1 \trianglelefteq V_2$ to hold in this case when, given $S_1 \trianglelefteq S_2$, the result of $S_2[\texttt{force}\ V_2]$ is approximated by $S_1[\texttt{force}\ V_1]$. To incorporate the indices, we have to quantify over $j \leq i$ in this definition because we need to know that the values are related in all futures, including ones where some other part of the term has been reduced (consuming some steps). Technically, this is crucial for making sure the relation is downward-closed. This is known as the *orthogonal* of the relation, and one advantage of the CBPV language is that it makes the use of orthogonality *explicit* in the type structure, analogous to the benefits of using Nakano's *later* modality [54] for step indexing (which we ironically do not do).

Next, we define when two *stacks* are related. First, we define the relation only for two "closed" stacks, which both have the same type of their hole $\underline{B}$ and both have "output" the observation type $\underline{F}2$. The reason is that in evaluating a program M, steps always occur as $S[N] \mapsto^* S[N']$ where S is a stack of this form. An intuition is that for negative types, two stacks are related when they start with the same elimination form and the remainder of the stacks are related. For ν, we handle the step indices in the same way as for μ. For $\underline{F}A$, a stack $S[\bullet : \underline{F}A]$ is strict in its input and waits for its input to evaluate down to a value $\texttt{ret}\ V$, so two stacks with $\underline{F}A$ holes are related when in any future world, they produce related behavior when given related values.

We note that in the CBV restriction of CBPV, the function type is given by $U(A \to \underline{F}A')$ and the logical relation we have presented reconstructs the usual definition that involves a double orthogonal.

Note that the definition is well-founded using the lexicographic ordering on (i, A) and $(i, \underline{B})$: either the type reduces and the index stays the same or the index reduces. We extend the definition to contexts to *closing substitutions* pointwise: two closing substitutions for Γ are related at i if they are related at i for each $x : A \in \Gamma$.

The logical preorder for open terms is defined as usual by quantifying over all related closing substitutions, but also over all stacks to the observation type $\underline{F}(1 + 1)$:

Definition 171 (Logical Preorder)**.** For a divergence preorder $\trianglelefteq$, its step-indexed logical preorder is

1. $\Gamma \vDash M_1 \trianglelefteq_i^{\log} M_2 \in \underline{B}$ iff for every $\gamma_1 \trianglelefteq_{\Gamma,i}^{\log} \gamma_2$ and $S_1 \trianglelefteq_{\underline{B},i}^{\log} S_2$,
$$S_1[M_1[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M_2[\gamma_2]]).$$

2. $\Gamma \vDash V_1 \trianglelefteq_i^{\log} V_2 \in A$ iff for every $\gamma_1 \trianglelefteq_{\Gamma,i}^{\log} \gamma_2$,
$$V_1[\gamma_1] \trianglelefteq_{A,i}^{\log} V_2[\gamma_2].$$

3. $\Gamma \mid \underline{B} \vDash S_1 \trianglelefteq_i^{\log} S_2 \in \underline{B}'$ iff for every $\gamma_1 \trianglelefteq_{\Gamma,i}^{\log} \gamma_2$ and $S_1' \trianglelefteq_{\underline{B}',i}^{\log} S_2'$,
$$S_1'[S_1[\gamma_1]] \trianglelefteq_{\underline{B},i}^{\log} S_2'[S_2[\gamma_2]]).$$

We next want to prove that the logical preorder is a congruence relation, i.e., the fundamental lemma of the logical relation. This requires the easy lemma, that the relation on closed terms and stacks is downward closed.

Lemma 172 (Logical Relation Downward Closure)**.** *For any type T, if $j \leq i$ then $\trianglelefteq_{T,i}^{log} \subseteq \trianglelefteq_{T,j}^{log}$*

Next, we show the fundamental theorem:

Theorem 173 (Logical Preorder is a Congruence)**.** *For any divergence preorder, the logical preorder $E \trianglelefteq_i^{log} E'$ is a congruence relation, i.e., it is closed under applying any value/term/stack constructors to both sides.*

Proof. For each congruence rule
$$\frac{\Gamma \mid \Delta \vdash E_1 \sqsubseteq E_1' : T_1 \cdots}{\Gamma' \mid \Delta' \vdash E_c \sqsubseteq E_c' : T_c}$$
we prove for every $i \in \mathbb{N}$ the validity of the rule
$$\frac{\Gamma \mid \Delta \vDash E_1 \trianglelefteq_i^{\log} E_1' \in T_1 \cdots}{\Gamma \mid \Delta \vDash E_c \trianglelefteq_i^{\log} E_c' \in T_c}$$

1. $\Gamma, x : A, \Gamma' \vDash x \trianglelefteq_i^{\log} x \in A$. Given $\gamma_1 \trianglelefteq_{\Gamma,x:A,\Gamma',i}^{\log} \gamma_2$, then by definition $\gamma_1(x) \trianglelefteq_{A,i}^{\log} \gamma_2(x)$.

2. $\Gamma \vDash \mho \trianglelefteq_i^{\log} \mho \in \underline{B}$ We need to show $S_1[\mho] \trianglelefteq^i \text{result}(S_2[\mho])$. By anti-reduction and strictness of stacks, it is sufficient to show $\mho \trianglelefteq_i^{\log} \mho$. If $i = 0$ there is nothing to show, otherwise, it follows by reflexivity of $\trianglelefteq$.

3. $$\frac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A \qquad \Gamma, x : A \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}}{\Gamma \vDash \texttt{let } x = V; M \trianglelefteq_i^{\log} \texttt{let } x = V'; M' \in \underline{B}}$$
Each side takes a 0-cost step, so by anti-reduction, this reduces to
$$S_1[M[\gamma_1, V/x]] \trianglelefteq^i \text{result}(S_2[M'[\gamma_2, V'/x]])$$
which follows by the assumption $\Gamma, x : A \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}$

4. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in 0}{\Gamma \vDash \mathtt{abort}\ V \preceq_i^{\log} \mathtt{abort}\ V' \in \underline{B}}$. By assumption, we get $V[\gamma_1] \preceq_{0,i}^{\log} V'[\gamma_2]$, but this is a contradiction.

5. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in A_1}{\Gamma \vDash \mathtt{inl}\ V \preceq_i^{\log} \mathtt{inl}\ V' \in A_1 + A_2}$. Direct from assumption, rule for sums.

6. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in A_2}{\Gamma \vDash \mathtt{inr}\ V \preceq_i^{\log} \mathtt{inr}\ V' \in A_1 + A_2}$ Direct from assumption, rule for sums.

7. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vDash M_1 \preceq_i^{\log} M_1' \in \underline{B} \qquad \Gamma, x_2 : A_2 \vDash M_2 \preceq_i^{\log} M_2' \in \underline{B}}{\Gamma \vDash \mathtt{case}\ V\{x_1.M_1 \mid x_2.M_2\} \preceq_i^{\log} \mathtt{case}\ V'\{x_1.M_1' \mid x_2.M_2'\} \in \underline{B}}$
By case analysis of $V[\gamma_1] \preceq_i^{\log} V'[\gamma_2]$.
   a) If $V[\gamma_1] = \mathtt{inl}\ V_1, V'[\gamma_2] = \mathtt{inl}\ V_1'$ with $V_1 \preceq_{A_1,i}^{\log} V_1'$, then taking 0 steps, by anti-reduction the problem reduces to
   $$S_1[M_1[\gamma_1, V_1/x_1]] \preceq^i \text{result}(S_1[M_1[\gamma_1, V_1/x_1]])$$
   which follows by assumption.
   b) For `inr` , the same argument.

8. $\Gamma \vDash () \preceq_i^{\log} () \in 1$ Immediate by unit rule.

9. $\dfrac{\Gamma \vDash V_1 \preceq_i^{\log} V_1' \in A_1 \qquad \Gamma \vDash V_2 \preceq_i^{\log} V_2' \in A_2}{\Gamma \vDash (V_1, V_2) \preceq_i^{\log} (V_1', V_2') \in A_1 \times A_2}$ Immediate by pair rule.

10. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in A_1 \times A_2 \qquad \Gamma, x : A_1, y : A_2 \vDash M \preceq_i^{\log} M' \in \underline{B}}{\Gamma \vDash \mathtt{let}\ (x,y) = V; M \preceq_i^{\log} \mathtt{let}\ (x,y) = V'; M' \in \underline{B}}$ By $V \preceq_{A_1 \times A_2, i}^{\log} V'$, we know $V[\gamma_1] = (V_1, V_2)$ and $V'[\gamma_2] = (V_1', V_2')$ with $V_1 \preceq_{A_1,i}^{\log} V_1'$ and $V_2 \preceq_{A_2,i}^{\log} V_2'$. Then by anti-reduction, the problem reduces to
   $$S_1[M[\gamma_1, V_1/x, V_2/y]] \preceq^i \text{result}(S_1[M'[\gamma_1, V_1'/x, V_2'/y]])$$
   which follows by assumption.

11. $\dfrac{\Gamma \vDash V \preceq_i^{\log} V' \in A[\mu X.A/X]}{\Gamma \vDash \mathtt{roll}_{\mu X.A}\ V \preceq_i^{\log} \mathtt{roll}_{\mu X.A}\ V' \in \mu X.A}$ If $i = 0$, we're done. Otherwise $i = j+1$, and our assumption is that $V[\gamma_1] \preceq_{A[\mu X.A/X],j+1}^{\log} V'[\gamma_2]$ and we need to show that $\mathtt{roll}\ V[\gamma_1] \preceq_{\mu X.A,j+1}^{\log} \mathtt{roll}\ V'[\gamma_2]$. By definition, we need to show $V[\gamma_1] \preceq_{A[\mu X.A/X],j}^{\log} V'[\gamma_2]$, which follows by downward-closure.

12. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in \mu X.A \qquad \Gamma, x : A[\mu X.A/X] \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}}{\Gamma \vDash \texttt{unroll}\ V\ \texttt{to roll}\ x.M \trianglelefteq_i^{\log} \texttt{unroll}\ V'\ \texttt{to roll}\ x.M' \in \underline{B}}$ If $i = 0$, then by triviality at 0, we're done. Otherwise, $V[\gamma_1] \trianglelefteq^{\log}_{\mu X.A, j+1} V'[\gamma_2]$ so $V[\gamma_1] = \texttt{roll}\ V_\mu, V'[\gamma_2] = \texttt{roll}\ V'_\mu$ with $V_\mu \trianglelefteq^{\log}_{A[\mu X.A/X], j} V'_\mu$. Then each side takes 1 step, so by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1, V_\mu/x]] \trianglelefteq^j \text{result}(S_2[M'[\gamma_2, V'_\mu/x]])$$

which follows by assumption and downward closure of the stack, value relations.

13. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}}{\Gamma \vDash \texttt{thunk}\ M \trianglelefteq_i^{\log} \texttt{thunk}\ M' \in U\underline{B}}$. We need to show $\texttt{thunk}\ M[\gamma_1] \trianglelefteq^{\log}_{U\underline{B}, i}$ $\texttt{thunk}\ M'[\gamma_2]$, so let $S_1 \trianglelefteq^{\log}_{\underline{B}, j} S_2$ for some $j \leq i$, and we need to show

$$S_1[\texttt{force thunk}\ M_1[\gamma_1]] \trianglelefteq^j \text{result}(S_2[\texttt{force thunk}\ M_2[\gamma_2]])$$

Then each side reduces in a 0-cost step and it is sufficient to show

$$S_1[M_1[\gamma_1]] \trianglelefteq^j \text{result}(S_2[M_2[\gamma_2]])$$

Which follows by downward-closure for terms and substitutions.

14. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in U\underline{B}}{\Gamma \vDash \texttt{force}\ V \trianglelefteq_i^{\log} \texttt{force}\ V' \in \underline{B}}$.
We need to show $S_1[\texttt{force}\ V[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{force}\ V'[\gamma_2]])$, which follows by the definition of $V[\gamma_1] \trianglelefteq^{\log}_{U\underline{B}, i} V'[\gamma_2]$.

15. $\dfrac{\Gamma \vDash V \trianglelefteq_i^{\log} V' \in A}{\Gamma \vDash \texttt{ret}\ V \trianglelefteq_i^{\log} \texttt{ret}\ V' \in \underline{F}A}$
We need to show $S_1[\texttt{ret}\ V[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{ret}\ V'[\gamma_2]])$, which follows by the orthogonality definition of $S_1 \trianglelefteq^{\log}_{\underline{F}A, i} S_2$.

16. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{F}A \qquad \Gamma, x : A \vDash N \trianglelefteq_i^{\log} N' \in \underline{B}}{\Gamma \vDash x \leftarrow M; N \trianglelefteq_i^{\log} x \leftarrow M'; N' \in \underline{B}}$.
We need to show $x \leftarrow M[\gamma_1]; N[\gamma_2] \trianglelefteq^i \text{result}(x \leftarrow M'[\gamma_2]; N'[\gamma_2])$. By $M \trianglelefteq_i^{\log} M' \in \underline{F}A$, it is sufficient to show that

$$x \leftarrow \bullet; N[\gamma_1] \trianglelefteq^{\log}_{\underline{F}A, i} x \leftarrow \bullet; N'[\gamma_2]$$

So let $j \leq i$ and $V \trianglelefteq^{\log}_{A, j} V'$, then we need to show

$$x \leftarrow \texttt{ret}\ V; N[\gamma_1] \trianglelefteq^{\log}_{\underline{F}A, j} x \leftarrow \texttt{ret}\ V'; N'[\gamma_2]$$

By anti-reduction, it is sufficient to show

$$N[\gamma_1, V/x] \trianglelefteq^j \text{result}(N'[\gamma_2, V'/x])$$

which follows by anti-reduction for $\gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2$ and $N \trianglelefteq^{\log}_i N'$.

17. $\dfrac{\Gamma, x : A \vDash M \trianglelefteq^{\log}_i M' \in \underline{B}}{\Gamma \vDash \lambda x : A.M \trianglelefteq^{\log}_i \lambda x : A.M' \in A \to \underline{B}}$ We need to show

$$S_1[\lambda x : A.M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\lambda x : A.M'[\gamma_2]]).$$

By $S_1 \trianglelefteq^{\log}_{A\to\underline{B},i} S_2$, we know $S_1 = S_1'[\bullet V_1]$, $S_2 = S_2'[\bullet V_2]$ with $S_1' \trianglelefteq^{\log}_{\underline{B},i} S_2'$ and $V_1 \trianglelefteq^{\log}_{A,i} V_2$. Then by anti-reduction it is sufficient to show

$$S_1'[M[\gamma_1, V_1/x]] \trianglelefteq^i \text{result}(S_2'[M'[\gamma_2, V_2/x]])$$

which follows by $M \trianglelefteq^{\log}_i M'$.

18. $\dfrac{\Gamma \vDash M \trianglelefteq^{\log}_i M' \in A \to \underline{B} \qquad \Gamma \vDash V \trianglelefteq^{\log}_i V' \in A}{\Gamma \vDash M\, V \trianglelefteq^{\log}_i M'\, V' \in \underline{B}}$ We need to show

$$S_1[M[\gamma_1]\, V[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M'[\gamma_2]\, V'[\gamma_2]])$$

so by $M \trianglelefteq^{\log}_i M'$ it is sufficient to show $S_1[\bullet V[\gamma_1]] \trianglelefteq^{\log}_{A\to\underline{B},i} S_2[\bullet V'[\gamma_2]]$ which follows by definition and assumption that $V \trianglelefteq^{\log}_i V'$.

19. $\Gamma \vdash \{\} : \top$ We assume we are given $S_1 \trianglelefteq^{\log}_{\top,i} S_2$, but this is a contradiction.

20. $\dfrac{\Gamma \vDash M_1 \trianglelefteq^{\log}_i M_1' \in \underline{B}_1 \qquad \Gamma \vDash M_2 \trianglelefteq^{\log}_i M_2' \in \underline{B}_2}{\Gamma \vDash (M_1, M_2) \trianglelefteq^{\log}_i (M_1', M_2') \in \underline{B}_1 \,\&\, \underline{B}_2}$ We need to show

$$S_1[(M_1[\gamma_1], M_2[\gamma_1])] \trianglelefteq^i \text{result}(S_2[(M_1'[\gamma_1], M_2'[\gamma_2])]).$$

We proceed by case analysis of $S_1 \trianglelefteq^{\log}_{\underline{B}_1\&\underline{B}_2,i} S_2$

a) In the first possibility $S_1 = S_1'[\pi\bullet], S_2 = S_2'[\pi\bullet]$ and $S_1' \trianglelefteq^{\log}_{\underline{B}_1,i} S_2'$. Then by anti-reduction, it is sufficient to show

$$S_1'[M_1[\gamma_1]] \trianglelefteq^i \text{result}(S_2'[M_1'[\gamma_2]])$$

which follows by $M_1 \trianglelefteq^{\log}_i M_1'$.

b) Same as previous case.

21. $\dfrac{\Gamma \vDash M \trianglelefteq^{\log}_i M' \in \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vDash \pi M \trianglelefteq^{\log}_i \pi M' \in \underline{B}_1}$ We need to show $S_1[\pi M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\pi M'[\gamma_2]])$, which follows by $S_1[\pi\bullet] \trianglelefteq^{\log}_{\underline{B}_1\&\underline{B}_2,i} S_2[\pi\bullet]$ and $M \trianglelefteq^{\log}_i M'$.

22. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vDash \pi' M \trianglelefteq_i^{\log} \pi' M' \in \underline{B}_2}$ Similar to previous case.

23. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}{\Gamma \vDash \mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M \trianglelefteq_i^{\log} \mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M' \in \nu \underline{Y}.\underline{B}}$ We need to show that

$$S_1[\mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2[\mathtt{roll}_{\nu \underline{Y}.\underline{B}}\ M'[\gamma_2]])$$

If $i = 0$, we invoke triviality at 0. Otherwise, $i = j + 1$ and we know by $S_1 \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B}, j+1} S_2$ that $S_1 = S_1'[\mathtt{unroll}\ \bullet]$ and $S_2 = S_2'[\mathtt{unroll}\ \bullet]$ with $S_1' \trianglelefteq^{\log}_{\underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}], j} S_2'$, so by anti-reduction it is sufficient to show

$$S_1'[M[\gamma_1]] \trianglelefteq^i \mathrm{result}(S_2'[M'[\gamma_2]])$$

which follows by $M \trianglelefteq_i^{\log} M'$ and downward-closure.

24. $\dfrac{\Gamma \vDash M \trianglelefteq_i^{\log} M' \in \nu \underline{Y}.\underline{B}}{\Gamma \vDash \mathtt{unroll}\ M \trianglelefteq_i^{\log} \mathtt{unroll}\ M' \in \underline{B}[\nu \underline{Y}.\underline{B}/\underline{Y}]}$ We need to show

$$S_1[\mathtt{unroll}\ M] \trianglelefteq^i \mathrm{result}(S_2[\mathtt{unroll}\ M']),$$

which follows because $S_1[\mathtt{unroll}\ \bullet] \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B}, i} S_2[\mathtt{unroll}\ \bullet]$ and $M \trianglelefteq_i^{\log} M'$.

□

As a direct consequence we get the reflexivity of the relation:

Corollary 174 (Reflexivity)**.** *For any* $\Gamma \vdash M : \underline{B}$*, and* $i \in \mathbb{N}$, $\Gamma \vDash M \trianglelefteq_i^{log} M \in \underline{B}$.

Therefore we have the following *strengthening* of the progress-and-preservation type soundness theorem: because $\trianglelefteq^i$ only counts unrolling steps, terms that never use μ or ν types (for example) are guaranteed to terminate.

Corollary 175 (Unary LR)**.** *For every program* $\cdot \vdash M : \underline{F}2$ *and* $i \in \mathbb{N}$, $M \trianglelefteq^i \mathit{result}(M)$

Proof. By reflexivity, $\cdot \vDash M \trianglelefteq^i M \in \underline{F}2$ and by definition $\bullet \trianglelefteq^{\log}_{\underline{F}2, i} \bullet$, so unrolling definitions we get $M \trianglelefteq^i \mathrm{result}(M)$. □

Using reflexivity, we prove that the indexed relation between terms and results recovers the original preorder in the limit as $i \to \omega$. We write $\trianglelefteq^\omega$ to mean the relation holds for every i, i.e., $\trianglelefteq^\omega = \bigcap_{i \in \mathbb{N}} \trianglelefteq^i$.

Corollary 176 (Limit Lemma)**.** *For any divergence preorder* $\trianglelefteq$*,* $\mathit{result}(M) \trianglelefteq R$ *iff* $M \trianglelefteq^\omega R$.

Proof. Two cases

1. If $\text{result}(M) \trianglelefteq R$ then we need to show for every $i \in \mathbb{N}$, $M \trianglelefteq^i R$. By the unary model lemma, $M \trianglelefteq^i \text{result}(M)$, so the result follows by the module Lemma 166.

2. If $M \trianglelefteq^i R$ for every i, then there are two possibilities: M is always related to R because it takes i steps, or at some point M terminates.

   a) If $M \mapsto^i M_i$ for every $i \in \mathbb{N}$, then $\text{result}(M) = \Omega$, so $\text{result}(M) \trianglelefteq R$ because $\trianglelefteq$ is a divergence preorder.

   b) Otherwise there exists some $i \in \mathbb{M}$ such that $M \mapsto^i \text{result}(M)$, so it follows by the module Lemma 166.

□

Corollary 177 (Logical implies Contextual)**.** *If* $\Gamma \vDash E \trianglelefteq^{log}_{\omega} E' \in \underline{B}$ *then* $\Gamma \vDash E \trianglelefteq^{ctx} E' \in \underline{B}$.

Proof. Let C be a closing context. By congruence, $C[M] \trianglelefteq^{\log}_{\omega} C[N]$, so using empty environment and stack, $C[M] \trianglelefteq^{\omega} \text{result}(C[N])$ and by the limit lemma, we have $\text{result}(C[M]) \trianglelefteq \text{result}(C[N])$. □

This establishes that our logical relation can prove graduality, so it only remains to show that our *inequational theory* implies our logical relation. Having already validated the congruence rules and reflexivity, we validate the remaining rules of transitivity, error, substitution, and $\beta\eta$ for each type constructor. Other than the $\mho \sqsubseteq M$ rule, all of these hold for any divergence preorder.

For transitivity, with the unary model and limiting lemmas in hand, we can prove that all of our logical relations (open and closed) are transitive in the limit. To do this, we first prove the following kind of "quantitative" transitivity lemma, and then transitivity in the limit is a consequence.

Lemma 178 (Logical Relation is Quantitatively Transitive)**.**

1. *If* $V_1 \trianglelefteq^{log}_{A,i} V_2$ *and* $V_2 \trianglelefteq^{log}_{A,\omega} V_3$*, then* $V_1 \trianglelefteq^{log}_{A,i} V_3$

2. *If* $S_1 \trianglelefteq^{log}_{\underline{B},i} S_2$ *and* $S_2 \trianglelefteq^{log}_{\underline{B},\omega} S_3$*, then* $S_1 \trianglelefteq^{log}_{\underline{B},i} S_3$

Proof. Proof is by mutual lexicographic induction on the pair (i, A) or $(i, \underline{B})$. All cases are straightforward uses of the inductive hypotheses except the shifts $U, \underline{F}$.

1. If $V_1 \trianglelefteq^{\log}_{U\underline{B},i} V_2$ and $V_2 \trianglelefteq^{\log}_{U\underline{B},\omega} V_3$, then we need to show that for any $S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2$ with $j \leq i$,

$$S_1[\texttt{force}\ V_1] \trianglelefteq^j \text{result}(S_2[\texttt{force}\ V_3])$$

By reflexivity, we know $S_2 \trianglelefteq^{\log}_{\underline{B},\omega} S_2$, so by assumption

$$S_2[\texttt{force}\ V_2] \trianglelefteq^{\omega} \text{result}(S_2[\texttt{force}\ V_3])$$

which by the limiting Lemma 176 is equivalent to

$$\text{result}(S_2[\texttt{force}\ V_2]) \trianglelefteq \text{result}(S_2[\texttt{force}\ V_3])$$

so then by the module Lemma 166, it is sufficient to show

$$S_1[\texttt{force}\ V_1] \trianglelefteq^{j} \text{result}(S_2[\texttt{force}\ V_2])$$

which holds by assumption.

2. If $S_1 \trianglelefteq^{\log}_{\underline{FA},i} S_2$ and $S_2 \trianglelefteq^{\log}_{\underline{FA},\omega} S_3$, then we need to show that for any $V_1 \trianglelefteq^{\log}_{j,A} V_2$ with $j \leq i$ that

$$S_1[\texttt{ret}\ V_1] \trianglelefteq^{j} \text{result}(S_3[\texttt{ret}\ V_2])$$

First by reflexivity, we know $V_2 \trianglelefteq^{\log}_{A,\omega} V_2$, so by assumption,

$$S_2[\texttt{ret}\ V_2] \trianglelefteq^{\omega} \text{result}(S_3[\texttt{ret}\ V_2])$$

Which by the limit Lemma 176 is equivalent to

$$\text{result}(S_2[\texttt{ret}\ V_2]) \trianglelefteq^{\omega} \text{result}(S_3[\texttt{ret}\ V_2])$$

So by the module Lemma 166, it is sufficient to show

$$S_1[\texttt{ret}\ V_1] \trianglelefteq^{j} \text{result}(S_2[\texttt{ret}\ V_2])$$

which holds by assumption.

□

Lemma 179 (Logical Relation is Quantitatively Transitive (Open Terms))**.**

1. *If* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_2$ *and* $\gamma_2 \trianglelefteq^{log}_{\Gamma,\omega} \gamma_3$*, then* $\gamma_1 \trianglelefteq^{log}_{\Gamma,i} \gamma_3$
2. *If* $\Gamma \vDash M_1 \trianglelefteq^{log}_{i} M_2 \in \underline{B}$ *and* $\Gamma \vDash M_2 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$*, then* $\Gamma \vDash M_1 \trianglelefteq^{log}_{i} M_3 \in \underline{B}$*.*
3. *If* $\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_2 \in A$ *and* $\Gamma \vDash V_2 \trianglelefteq^{log}_{\omega} V_3 \in A$*, then* $\Gamma \vDash V_1 \trianglelefteq^{log}_{i} V_3 \in A$*.*
4. *If* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{i} S_2 \in \underline{B}'$ *and* $\Gamma \mid \bullet : \underline{B} \vDash S_2 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$*, then* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{i} S_3 \in \underline{B}'$*.*

Proof. 1. By induction on the length of the context, follows from closed value case.

2. Assume $\gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2$ and $S_1 \trianglelefteq^{\log}_{\underline{B},i} S_2$. We need to show

   $$S_1[M_1[\gamma_1]] \trianglelefteq^{i} \text{result}(S_2[M_3[\gamma_2]])$$

   by reflexivity and assumption, we know

   $$S_2[M_2[\gamma_2]] \trianglelefteq^{\omega} \text{result}(S_2[M_3[\gamma_2]])$$

   and by limit Lemma 176, this is equivalent to

   $$\text{result}(S_2[M_2[\gamma_2]]) \trianglelefteq \text{result}(S_2[M_3[\gamma_2]])$$

   so by the module Lemma 166 it is sufficient to show

   $$S_1[M_1[\gamma_1]] \trianglelefteq^{i} \text{result}(S_2[M_2[\gamma_2]])$$

   which follows by assumption.

3. Assume $\gamma_1 \trianglelefteq^{\log}_{\Gamma,i} \gamma_2$. Then $V_1[\gamma_1] \trianglelefteq^{\log}_{A,i} V_2[\gamma_2]$ and by reflexivity $\gamma_2 \trianglelefteq^{\log}_{\Gamma,\omega} \gamma_2$ so $V_2[\gamma_2] \trianglelefteq^{\log}_{A,\omega} V_3[\gamma_2]$ so the result holds by the closed case.

4. Stack case is essentially the same as the value case.

□

Corollary 180 (Logical Relation is Transitive in the Limit). *1. If* $\Gamma \vDash M_1 \trianglelefteq^{log}_{\omega} M_2 \in \underline{B}$ *and* $\Gamma \vDash M_2 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$*, then* $\Gamma \vDash M_1 \trianglelefteq^{log}_{\omega} M_3 \in \underline{B}$.

2. *If* $\Gamma \vDash V_1 \trianglelefteq^{log}_{\omega} V_2 \in A$ *and* $\Gamma \vDash V_2 \trianglelefteq^{log}_{\omega} V_3 \in A$*, then* $\Gamma \vDash V_1 \trianglelefteq^{log}_{\omega} V_3 \in A$.

3. *If* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{\omega} S_2 \in \underline{B}'$ *and* $\Gamma \mid \bullet : \underline{B} \vDash S_2 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$*, then* $\Gamma \mid \bullet : \underline{B} \vDash S_1 \trianglelefteq^{log}_{\omega} S_3 \in \underline{B}'$.

Next, we verify the β, η equivalences hold as orderings each way.

Lemma 181 (β, η). *For any divergence preorder, the* β, η *laws are valid for* $\trianglelefteq^{log}_{\omega}$

Proof. The β rules for all cases except recursive types are direct from anti-reduction.

1. $\mu X.A - \beta$:

   a) We need to show

$$S_1[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_1]\ \texttt{to roll}\ x.M[\gamma_1]] \trianglelefteq^{\log}_{i} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   The left side takes 1 step to $S_1[M[\gamma_1, V[\gamma_1]/x]]$ and we know

   $$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq^{\log}_{i} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   by assumption and reflexivity, so by anti-reduction we have

$$S_1[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_1]\ \texttt{to roll}\ x.M[\gamma_1]] \trianglelefteq^{\log}_{i+1} \text{result}(S_2[M[\gamma_2, V[\gamma_2]/x]])$$

   so the result follows by downward-closure.

b) For the other direction we need to show

$$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq_i^{\log} \text{result}(S_2[\texttt{unroll roll}_{\mu X.A}\ V[\gamma_2]\ \texttt{to roll}\ x.M[\gamma_2]])$$

Since results are invariant under steps, this is the same as

$$S_1[M[\gamma_1, V[\gamma_1]/x]] \trianglelefteq_i^{\log} \text{result}(S_2[M[\gamma_2, V[\gamma_2/x]]])$$

which follows by reflexivity and assumptions about the stacks and substitutions.

2. $\mu X.A - \eta$:

a) We need to show for any $\Gamma, x : \mu X.A \vdash M : \underline{B}$, and appropriate substitutions and stacks,

$$S_1[\texttt{unroll roll}_{\mu X.A}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_1]] \trianglelefteq_i^{\log} \text{result}(S_2[M[\gamma_2]])$$

By assumption, $\gamma_1(x) \trianglelefteq_{\mu X.A,i}^{\log} \gamma_2(x)$, so we know

$$\gamma_1(x) = \texttt{roll}_{\mu X.A}\ V_1$$

and

$$\gamma_2(x) = \texttt{roll}_{\mu X.A}\ V_2$$

so the left side takes a step:

$$\begin{aligned} S_1[\texttt{unroll roll}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}\ y/x][\gamma_1]] &\mapsto^1 S_1[M[\texttt{roll}\ y/x][\gamma_1 \\ &= S_1[M[\texttt{roll}\ V_1/x][\gamma_1 \\ &= S_1[M[\gamma_1]] \end{aligned}$$

and by reflexivity and assumptions we know

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \text{result}(S_2[M[\gamma_2]])$$

so by anti-reduction we know

$$S_1[\texttt{unroll roll}_{\mu X.A}\ \gamma_1(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_1]] \trianglelefteq_{i+1}^{\log} \text{result}(S_2[M[\gamma_2]])$$

so the result follows by downward closure.

b) Similarly, to show

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \text{result}(S_2[\texttt{unroll roll}_{\mu X.A}\ \gamma_2(x)\ \texttt{to roll}\ y.M[\texttt{roll}_{\mu X.A}\ y/x][\gamma_2]])$$

by the same reasoning as above, $\gamma_2(x) = \texttt{roll}_{\mu X.A}\ V_2$, so because result is invariant under reduction we need to show

$$S_1[M[\gamma_1]] \trianglelefteq_i^{\log} \text{result}(S_2[M[\gamma_2]])$$

which follows by assumption and reflexivity.

3. $\nu\underline{Y}.\underline{B} - \beta$
   a) We need to show

      $$S_1[\texttt{unroll roll}_{\nu\underline{Y}.\underline{B}}\ M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      By the operational semantics,

      $$S_1[\texttt{unroll roll}_{\nu\underline{Y}.\underline{B}}\ M[\gamma_1]] \mapsto^1 S_1[M[\gamma_1]]$$

      and by reflexivity and assumptions

      $$S_1[M[\gamma_1]] \trianglelefteq^i S_2[M[\gamma_2]]$$

      so the result follows by anti-reduction and downward closure.
   b) We need to show

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{unroll roll}_{\nu\underline{Y}.\underline{B}}\ M[\gamma_2]])$$

      By the operational semantics and invariance of result under reduction this is equivalent to

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      which follows by assumption.

4. $\nu\underline{Y}.\underline{B} - \eta$
   a) We need to show

      $$S_1[\texttt{roll unroll}\ M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      by assumption, $S_1 \trianglelefteq^{\log}_{\nu\underline{Y}.\underline{B},i} S_2$, so

      $$S_1 = S_1'[\texttt{unroll } \bullet]$$

      and therefore the left side reduces:

      $$\begin{aligned} S_1[\texttt{roll unroll}\ M[\gamma_1]] &= S_1'[\texttt{unroll roll unroll}\ M[\gamma_1]] \\ &\mapsto^1 S_1'[\texttt{unroll}\ M[\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

      and by assumption and reflexivity,

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      so the result holds by anti-reduction and downward-closure.
   b) Similarly, we need to show

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[\texttt{roll unroll}\ M[\gamma_2]])$$

as above, $S_1 \trianglelefteq^{\log}_{\nu \underline{Y}.\underline{B},i} S_2$, so we know

$$S_2 = S_2'[\texttt{unroll}\ \bullet]$$

so

$$\text{result}(S_2[\texttt{roll unroll}\ M[\gamma_2]]) = \text{result}(S_2[M[\gamma_2]])$$

and the result follows by reflexivity, anti-reduction and downward closure.

5. 0η Let $\Gamma, x : 0 \vdash M : \underline{B}$.
   a) We need to show

$$S_1[\texttt{absurd}\ \gamma_1(x)] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   By assumption $\gamma_1(x) \trianglelefteq^{\log}_{0,i} \gamma_2(x)$ but this is a contradiction
   b) Other direction is the same contradiction.

6. $+\eta$. Let $\Gamma, x : A_1 + A_2 \vdash M : \underline{B}$
   a) We need to show

$$S_1[\texttt{case}\ \gamma_1(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_1] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_1]\}] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

   by assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1+A_2,i} \gamma_2(x)$, so either it's an `inl` or *inr*. The cases are symmetric so assume $\gamma_1(x) = \texttt{inl}\ V_1$. Then

$$\begin{aligned} & S_1[\texttt{case}\ \gamma_1(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_1] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_1]\}] \\ = {} & S_1[\texttt{case}\ (\texttt{inl}\ V_1)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_1] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_1]\}] \\ \mapsto^0 {} & S_1[M[\texttt{inl}\ V_1/x][\gamma_1]] \\ = {} & S_1[M[\gamma_1]] \end{aligned}$$

   and so by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^i S_2[M[\gamma_2]]$$

   which follows by reflexivity and assumptions.
   b) Similarly, We need to show

$$\text{result}(S_1[M[\gamma_1]]) \trianglelefteq^i \text{result}(S_2[\texttt{case}\ \gamma_2(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_2] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_2]\}])$$

   and by assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1+A_2,i} \gamma_2(x)$, so either it's an `inl` or *inr*. The cases are symmetric so assume $\gamma_2(x) = \texttt{inl}\ V_2$. Then

$$S_2[\texttt{case}\ \gamma_2(x)\{x_1.M[\texttt{inl}\ x_1/x][\gamma_2] \mid x_2.M[\texttt{inr}\ x_2/x][\gamma_2]\}] \mapsto^0 S_2[M[\gamma_2]]$$

   So the result holds by invariance of result under reduction, reflexivity and assumptions.

7. 1η Let $\Gamma, x : 1 \vdash M : \underline{B}$
   a) We need to show

      $$S_1[M[()/x][\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      By assumption $\gamma_1(x) \trianglelefteq^{\log}_{1,i} \gamma_2(x)$ so $\gamma_1(x) = ()$, so this is equivalent to

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      which follows by reflexivity, assumption.
   b) Opposite case is similar.
8. $\times\eta$ Let $\Gamma, x : A_1 \times A_2 \vdash M : \underline{B}$
   a) We need to show

      $$S_1[\texttt{let } (x_1, y_1) = x; M[(x_1, y_1)/x][\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      By assumption $\gamma_1(x) \trianglelefteq^{\log}_{A_1 \times A_2, i} \gamma_2(x)$, so $\gamma_1(x) = (V_1, V_2)$, so

      $$\begin{aligned} S_1[\texttt{let } (x_1, y_1) = x; M[(x_1, y_1)/x][\gamma_1]] &= S_1[\texttt{let } (x_1, y_1) = (V_1, V_2); M[(x_1, y_1)/x][\gamma_1]] \\ &\mapsto^0 S_1[M[(V_1, V_2)/x][\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

      So by anti-reduction it is sufficient to show

      $$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

      which follows by reflexivity, assumption.
   b) Opposite case is similar.
9. $U\eta$ Let $\Gamma \vdash V : U\underline{B}$
   a) We need to show that

      $$\texttt{thunk force } V[\gamma_1] \trianglelefteq^{\log}_{U\underline{B},i} V[\gamma_2]$$

      So assume $S_1 \trianglelefteq^{\log}_{\underline{B},j} S_2$ for some $j \leq i$, then we need to show

      $$S_1[\texttt{force thunk force } V[\gamma_1]] \trianglelefteq^j \text{result}(S_2[\texttt{force } V[\gamma_2]])$$

      The left side takes a step:

      $$S_1[\texttt{force thunk force } V[\gamma_1]] \mapsto^0 S_1[\texttt{force } V[\gamma_1]]$$

      so by anti-reduction it is sufficient to show

      $$S_1[\texttt{force } V[\gamma_1]] \trianglelefteq^j \text{result}(S_2[\texttt{force } V[\gamma_2]])$$

      which follows by assumption.

b) Opposite case is similar.

10. $F\eta$

a) We need to show that given $S_1 \trianglelefteq^{\log}_{\underline{F}A,i} S_2$,

$$S_1[x \leftarrow \bullet; \mathtt{ret}\ x] \trianglelefteq^{\log}_{\underline{F}A,i} S_2$$

So assume $V_1 \trianglelefteq^{\log}_{A,j} V_2$ for some $j \leq i$, then we need to show

$$S_1[\mathtt{ret}\ V_1 \leftarrow \bullet; \mathtt{ret}\ x] \trianglelefteq^{j} \mathrm{result}(S_2[\mathtt{ret}\ V_2])$$

The left side takes a step:

$$S_1[\mathtt{ret}\ V_1 \leftarrow \bullet; \mathtt{ret}\ x] \mapsto^0 S_1[\mathtt{ret}\ V_1]$$

so by anti-reduction it is sufficient to show

$$S_1[\mathtt{ret}\ V_1] \trianglelefteq^{j} \mathrm{result}(S_2[\mathtt{ret}\ V_2])$$

which follows by assumption

b) Opposite case is similar.

11. $\to \eta$ Let $\Gamma \vdash M : A \to \underline{B}$

a) We need to show

$$S_1[(\lambda x : A.M[\gamma_1]\, x)] \trianglelefteq^{i} \mathrm{result}(S_2[M[\gamma_2]])$$

by assumption that $S_1 \trianglelefteq^{\log}_{A\to\underline{B},i} S_2$, we know

$$S_1 = S_1'[\bullet\, V_1]$$

so the left side takes a step:

$$\begin{aligned} S_1[(\lambda x : A.M[\gamma_1]\, x)] &= S_1'[(\lambda x : A.M[\gamma_1]\, x)\, V_1] \\ &\mapsto^0 S_1'[M[\gamma_1]\, V_1] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

So by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^{i} \mathrm{result}(S_2[M[\gamma_2]])$$

which follows by reflexivity, assumption.

b) Opposite case is similar.

12. $\&\eta$ Let $\Gamma \vdash M : \underline{B}_1 \,\&\, \underline{B}_2$

a) We need to show

$$S_1[(\pi M[\gamma_1], \pi' M[\gamma_1])] \trianglelefteq^{i} \mathrm{result}(S_1[M[\gamma_2]])$$

by assumption, $S_1 \trianglelefteq^{\log}_{\underline{B}_1 \& \underline{B}_2, i} S_2$ so either it starts with a π or π' so assume that $S_1 = S_1'[\pi \bullet]$ (π' case is similar). Then the left side reduces

$$\begin{aligned} S_1[(\pi M[\gamma_1], \pi' M[\gamma_1])] &= S_1'[\pi(\pi M[\gamma_1], \pi' M[\gamma_1])] \\ &\mapsto^0 S_1'[\pi M[\gamma_1]] \\ &= S_1[M[\gamma_1]] \end{aligned}$$

So by anti-reduction it is sufficient to show

$$S_1[M[\gamma_1]] \trianglelefteq^i \text{result}(S_2[M[\gamma_2]])$$

which follows by reflexivity, assumption.

b) Opposite case is similar.

13. $\top\eta$ Let $\Gamma \vdash M : \top$

a) In either case, we assume we are given $S_1 \trianglelefteq^{\log}_{\top, i} S_2$, but this is a contradiction.

□

And that the logical relation behaves well is closed under substitution of related terms.

Lemma 182 (Substitution Principles)**.** *For any divergence preorder $\trianglelefteq$, the following are valid*

1. $$\frac{\Gamma \vDash V_1 \trianglelefteq^{log}_i V_2 \in A \qquad \Gamma, x : A \vDash V_1' \trianglelefteq^{log'}_{V\ 2} \in A'}{\Gamma \vDash V_1'[V_1/x] \trianglelefteq^{log'}_{V\ 2} [V_2/x] \in A'}$$

2. $$\frac{\Gamma \vDash V_1 \trianglelefteq^{log}_i V_2 \in A \qquad \Gamma, x : A \vDash M_1 \trianglelefteq^{log}_{M\ 2} \in \underline{B}}{\Gamma \vDash M_1[V_1/x] \trianglelefteq^{log}_{M\ 2} [V_2/x] \in \underline{B}}$$

Proof. We do the term case, the value case is similar. Given $\gamma_1 \trianglelefteq^{\log}_{\Gamma, i} \gamma_2$, we have $V_1[\gamma_1] \trianglelefteq^{\log}_{A, i} V_2[\gamma_2]$ so

$$\gamma_1, V_1[\gamma_1]/x \trianglelefteq^{\log}_{\Gamma, x:A, i} \gamma_2, V_2[\gamma_2]/x$$

and by associativity of substitution

$$M_1[V_1/x][\gamma_1] = M_1[\gamma_1, V_1[\gamma_1]/x]$$

and similarly for M_2, so if $S_1 \trianglelefteq^{\log}_{\underline{B}, i} S_2$ then

$$S_1[M_1[\gamma_1, V_1[\gamma_1]/x]] \trianglelefteq^i \text{result}(S_2[M_2[\gamma_2, V_2[\gamma_2]/x]])$$

□

For errors, the strictness axioms hold for any $\trianglelefteq$, but the axiom that $\mho$ is a least element is specific to the definitions of $\preceq\sqsubset, \sqsubseteq\succeq$

Lemma 183 (Error Rules)**.** *For any divergence preorder $\trianglelefteq$ and appropriately typed S, M,*

$$S[\mho] \trianglelefteq_\omega^{log} \mho \qquad \mho \trianglelefteq_\omega^{log} S[\mho] \qquad \mho \preceq\sqsubseteq_\omega^{log} M \qquad M \preceq\sqsupseteq_\omega^{log} \mho$$

Proof. 1. It is sufficient by the limit lemma to show $\text{result}(S[\mho]) \trianglelefteq \mho$ which holds by reflexivity because $S[\mho] \mapsto^0 \mho$.

2. We need to show $S[\mho] \preceq\sqsubseteq^i R$ for arbitrary R, so by the limit lemma it is sufficient to show $\mho \preceq\sqsubseteq R$, which is true by definition.

3. By the limit lemma it is sufficient to show $R \preceq\sqsupseteq \mho$ which is true by definition.

□

The lemmas we have proved cover all of the inequality rules of CBPV, so applying them with $\trianglelefteq$ chosen to be $\preceq\sqsubseteq$ and $\preceq\sqsupseteq$ gives

Lemma 184 ($\preceq\sqsubseteq$ and $\sqsubseteq\succeq$ are Models of CBPV)**.** *If $\Gamma \mid \Delta \vdash E \sqsubseteq E' : \underline{B}$ then $\Gamma \mid \Delta \vDash E \preceq\sqsubseteq^\omega E' \in \underline{B}$ and $\Gamma \mid \Delta \vDash E' \preceq\sqsupseteq^\omega E \in \underline{B}$.*

Because logical implies contextual equivalence, we can conclude with the main theorem:

Theorem 185 (Contextual Approximation/Equivalence Model CBPV)**.**
If $\Gamma \mid \Delta \vdash E \sqsubseteq E' : T$ then $\Gamma \mid \Delta \vDash E \sqsubseteq^{ctx} E' \in T$.
If $\Gamma \mid \Delta \vdash E \sqsupseteq\sqsubseteq E' : T$ then $\Gamma \mid \Delta \vDash E =^{ctx} E' \in T$.

Proof. For the first part, from Lemma 184, we have $E \preceq\sqsubseteq^\omega E'$ and $E' \preceq\sqsupseteq^\omega E$. By Lemma 177, we then have $E \preceq\sqsubseteq^{\text{ctx}} E'$ and $E' \preceq\sqsupseteq^{\text{ctx}} E$. Finally, by Lemma 164, $E \sqsubseteq^{\text{ctx}} E'$ iff $E \preceq\sqsubseteq^{\text{ctx}} E'$and $E((\preceq\sqsupseteq)^{\text{ctx}})^\circ E'$, so we have the result.

For the second part, applying the first part twice gives $E \sqsubseteq^{\text{ctx}} E'$ and $E' \sqsubseteq^{\text{ctx}} E$, and we concluded in Lemma 164 that this coincides with contextual equivalence. □

Part III

GRADUAL TYPING AND PARAMETRIC POLYMORPHISM

8

INTRODUCTION TO PART III

While simple types like sums, products and extensional functions have their cast structure completely determined by $\beta\eta$ equations, many programming features and their associated reasoning principles do not naturally fall into this pattern. In the following chapters we focus on the problem of developing a gradually typed language whose typing features include support for data abstraction using *parametric polymorphism*, i.e. universal and existential types. Our main design goal is to develop languages that satisfy both *graduality* and *parametricity*.

Parametric polymorphism, in the form of universal and existential types, allows for abstraction over types within a program. Universal types, written $\forall X.A$, allow for the definition of functions that can be used at many different types. Dually, existential types provide a simple model of a module system. A value of type $\exists X.A$ can be thought of as a module that exports a newly defined type X and then a value A that may include X that gives the interface to the type. Languages with parametric polymorphism provide very strong reasoning principles regarding data abstraction, formalized by the relational parametricity theorem [64].

The relational parametricity theorem captures the idea that an abstract type is truly opaque to its users: for instance, a consumer of a value of existential type $\exists X.A$ can only interact with X values using the capabilities provided by the interface type A. This allows programmers to use existential types to model abstract data types [50]. For instance, the existential type $\exists X.X \times (X \to X) \times (X \to \texttt{Int})$ represents the type of an abstract functional counter. The X represents the state, the first component of the tuple is the initial state, the second component is an increment function, and the final component reads out an observable integer value from the state. One obvious example implementation would use `Int` for X, 0 as the initial state, addition by 1 as the increment, and the identity function as the read-out. In a language with proper data abstraction, we should be able to guarantee that with this implementation, the read-out function should only ever produce positive numbers, because even though the type `Int` allows for negative numbers, the interface only enables the construction of positive numbers. This pattern of reasoning naturally generalizes to sophisticated data structure invariants such as balanced trees, sorted lists, etc.

In this part of the dissertation we will analyze to what extent parametric polymorphism and parametricity are compatible with gradual typing. First, in Chapter 9, we will show that under certain conditions

common in dynamically typed languages, enforcement of parametricity is not computable, and so no gradual typing scheme for parametricity can be developed that is not an over approximation. Then in Chapter 10, we analyze previous attempts to combine parametricity and gradual typing using a technique called *dynamic sealing* and develop a new language design with a novel syntax that makes dynamic sealing explicit and provides both data abstraction and graduality.

Chapter 10 is an expanded version of the paper New, Jamner, and Ahmed [57], with a simpler syntax for polymorphic function application. Chapter 9 is novel material.

9

GRADUAL TYPING & CURRY STYLE POLYMORPHISM

In this chapter we study the problem of correct dynamic enforcement of parametricity for *Curry*-style, i.e., *implicitly* polymorphic, languages, and show that it is not computable if the language features any non-trivial total functions on the dynamic type such as tag tests, instance checks or printing. By Curry-style polymorphism we mean languages with polymorphic function types but where this polymorphism is not present in the syntax of terms. For instance, in a Curry-style system the following identity function

$$\lambda x.x$$

can be given the polymorphic type $\forall X.X \to X$. By contrast, in a Church-style or explicitly polymorphic language, the polymorphic type variable would need to be explicitly quantified in the syntax, and a polymorphic identity function would be written

$$\Lambda X.\lambda x : X.x.$$

In the context of gradual typing, Curry-style is especially relevant because dynamically typed languages do not typically quantify over type variables, and so if the typed sublanguage has the same syntax as the dynamic language, then Curry-style is the most relevant.

9.1 INFORMAL PROOF

In this section we show that correct enforcement of Curry-style parametricity is incompatible with having a total, non-trivial predicate in the dynamically typed sublanguage. Note that total, non-trivial predicates are quite common in dynamically typed programming: testing if an input is an integer, if an input is an instance of a specific class, or printing an input to a string are all typical, benign features of dynamically typed languages that allow for the construction of such a predicate.

First, we present an informal argument. Let p be our predicate and V_t be an input such that $p\ V_t \mapsto^* \texttt{true}$ and V_f an input such that $p\ V_f \mapsto^* \texttt{false}$. Let N be an arbitrary closed dynamically typed program. We will show that it is decidable if $\texttt{let}\ y = N; \texttt{true} \cong \texttt{true}$. This would imply that we can determine in finite time if a term N diverges.

Construct the following program $Q_?[N]$:

$$Q_?[N] = \lambda^?x.\texttt{if}\ p(x)\ \texttt{then}\ (\texttt{let}\ y = N; \texttt{true})\ \texttt{else}\ \texttt{true}$$

$Q_?[N]$ is a dynamically typed function that checks if its input satisfies p and either runs N and then returns `true`, or just returns `true`. Next, define $Q_\forall$ to be that function cast to a polymorphic type of constant functions that return booleans:

$$Q_\forall[N] = Q_?[N] :: \forall X.X \to \texttt{Bool}$$

The free theorem for this type says that the input doesn't matter at all. For any $V_A : A$ and $V_B : B$, we should have

$$Q_\forall[N][A]V_A \cong Q_\forall[N]\ [A]\ V_B$$

Now, note that if $\texttt{let}\ y = N; \texttt{true} \cong \texttt{true}$, then $Q_?[N]$ is equivalent to a constant function $\lambda^? x.\texttt{true}$. Constant functions are clearly parametric so $Q_\forall[N]$ should also be equivalent to a function that always returns `true`. However, if $\texttt{let}\ y = N; \texttt{true} \ncong \texttt{true}$, then we can pass values to $Q_?[N]$ to have it perform each behavior: if we pass it V_t it will perform N's side-effects, but if passed V_f it will reduce to `true` with no observable side-effects.

By graduality, $Q_\forall[N]$ must approximate the behavior of $Q_?[N]$, so when passed V_t it must "error more than" $\texttt{let}\ y = N; \texttt{true}$ and if passed V_f it must error more than simply `true`. But by parametricity, its behavior must be the same in both cases since it cannot depend on its input so it must have a single behavior that errors more than `true` and also $\texttt{let}\ y = N; \texttt{true}$. Finally, the only valid behaviors that error more than `true` are Ω and `true` itself, so $Q_\forall[N]$ must always error.

So to summarize, to decide if $\texttt{let}\ y = N; \texttt{true} \cong \texttt{true}$, run the program:

$$Q_\forall[N]V_t$$

If $\texttt{let}\ y = N; \texttt{true} \cong \texttt{true}$ then this will reduce to `true` in finite time, otherwise it will reduce to Ω in finite time.

9.2 FORMALIZING THE ASSUMPTIONS OF THE PROOF

In order to formalize what the assumptions of the proof above are, we list some axioms that are sufficient to reproduce the proof. To keep things as generalizable as possible, we will axiomatize this based on an equivalence relation between closed programs $\cong$. This corresponds in our previous chapters to "semantic equivalence", for instance saying two terms are equivalent if they both terminate, both error or both diverge. But we leave this abstract to avoid making assumptions about what effects are present in the language. To be completely precise, some axioms will fix a call-by-value evaluation order, but the analysis should be easily modifiable to other evaluation orders.

First, we review some basic relevant notions of computability theory. Informally a predicate is *semidecidable* if we can determine in finite time

if something satisfies a predicate, but might not be able to determine in finite time if something does not satisfy a predicate. Formally, a predicate on strings Q is semidecidable if there exists a program in a Turing-equivalent language that accepts an input s if and only $Q(s)$ is true. On other inputs, the program may either diverge or reject the input. A predicate is *co-semidecidable* if its complement is semidecidable. A predicate Q is *decidable* when it is semidecidable and co-semidecidable. Equivalently, it can be correctly implemented as an always terminating program that accepts when inputs satisfy the predicate and rejects otherwise.

Now, the axioms we choose are as follows.

- The language is sufficiently strong to satisfy Rice's theorem: any total, computable, semantic predicate of terms in the language is trivial: always true or always false. Here by semantic we will mean that it respects $\cong$.
- Termination and error are finitary in that $M \cong \texttt{true}$ and $M \cong \mho$ are both semidecidable.
- Some basic call-by-value reductions
  - $\texttt{let}\ x = \texttt{true}; \texttt{true} \cong \texttt{true}$ and $\texttt{let}\ x = \mho; \texttt{true} \cong \mho$. In lazy languages, this could be replaced by an if-statement or any other strict function.
  - Call-by-value β reduction for dynamically typed functions. We need that for pure terms V.

    $$(\lambda^{?} x.M)\ V \cong M[V/x]$$

  - β reduction for if-statements:

    $$\texttt{if true then}\ M\ \texttt{else}\ N \cong M$$

    and

    $$\texttt{if false then}\ M\ \texttt{else}\ N \cong N$$

- There is a non-trivial total predicate on the dynamic type. I.e., a term in the language p such that for any dynamically typed input v, $p(v)$ reduces deterministically to either `true` or `false` and there is at least one value v_t such that $p(v_t)$ is `true` and at least one value v_f such that $p(v_f)$ is `false`.

  More formally, we say a term t is pure if for any u, $\texttt{let}\ y = t; u \cong u[t/y]$. First, we require that p is a total predicate in that for any pure t, $p(t)$ is pure. Then, we say that it is boolean in that

  $$\frac{x\,\text{dynamic} \vdash u}{\texttt{let}\ x = p(t); u \cong \texttt{if}\ p(t)\ \texttt{then}\ u[\texttt{true}/x]\ \texttt{else}\ u[\texttt{false}/x]} : A$$

And non-trivial in that there are pure terms V_t and V_f such that $p(V_t) \cong \texttt{true}$ and $p(V_f) \cong \texttt{false}$.

A typical example of such a predicate in a dynamic language would be a tag-testing function such as integer? in Racket, or checking if an object is an instance of a given class. In addition, any total, non-trivial function that prints values to an observable string can likely be used to construct such a predicate by checking if the first character of the string is equal to "a".

- To formalize graduality of the language, we have an ordering $\sqsubseteq$ on terms (possibly of different types). Think of this ordering not as the syntactic term precision ordering, but rather the semantic term ordering that term precision implies.
  1. The graduality ordering respects semantic equivalence.

$$\frac{M' \cong M \qquad M \sqsubseteq N \qquad N \cong N'}{M' \sqsubseteq N'}$$

  2. (Ground Value) If $M \ncong \texttt{true}$, and $N \sqsubseteq M$ and $N \sqsubseteq \texttt{true}$, then $N \cong \mho$.
  3. Polymorphic functions are considered more precise than (non-polymorphic) dynamically typed functions and polymorphic type instantiation is more precise than dynamically typed function application in that

$$\frac{M_\forall \sqsubseteq M_d : \forall X.X \to X \sqsubseteq ? \to ? \qquad N_X \sqsubseteq N_d : A \sqsubseteq ?}{M_\forall[A]\ N_X \sqsubseteq M_d\ N_d : A \sqsubseteq ?}$$

- To formalize parametricity, we ask only for a couple of consequences of parametricity for terms of type $M : \forall X.X \to \texttt{Bool}$. The first property tells us that all such functions are uniform, and the second gives us an example of a function that should successfully be castable to this type.
  1. Every term $M : \forall X.X \to \texttt{Bool}$ is uniform in that for any A and $V_A : A$ and $V_B : B$ that

$$M[A]V_A \cong M[B]V_B$$

  2. The constant function $\lambda x.\texttt{true}$ is parametric in that casting $\lceil \lambda x.\texttt{true} \rceil :: \forall X.X \to \texttt{Bool}$ results in a term satisfying

$$(\lceil \lambda x.\texttt{true} \rceil :: \forall X.X \to \texttt{Bool})[A]M \cong \texttt{let } y = M; \texttt{true}$$

Then we can show the following lemma which shows that running $Q_\forall\ [N]\ V_t$ is equivalent to testing N runs to $\texttt{true}$.

Lemma 186. *Define (as before)*

$$Q_?[N] = \lambda^?x.\texttt{if } p(x) \texttt{ then } (\texttt{let } y = N; \texttt{true}) \texttt{ else true}$$

and

$$Q_\forall[N] = Q_?[N] :: \forall X.X \to \texttt{Bool}$$

Then from our axioms we can show that,

$$Q_\forall[N]\ [?]V_t \cong \texttt{true} \quad \textit{if and only if} \quad \texttt{let } y = N; \texttt{true} \cong \texttt{true}$$

and

$$Q_\forall[N]\ [?]V_t \cong \mho \quad \textit{if and only if} \quad \texttt{let } y = N; \texttt{true} \ncong \texttt{true}$$

Proof. By excluded middle and since $\mho \ncong \texttt{true}$, it is sufficient to show the reverse cases only.

- Assume $\texttt{let } y = N; \texttt{true} \cong \texttt{true}$. We need to show that $Q_\forall[N]\ [?]\ V_t \cong \texttt{true}$.

  First, since $\texttt{let } y = N; \texttt{true} \cong \texttt{true}$,

  $$\begin{aligned}
  &Q_?[N] \\
  &= \lambda^?x.\texttt{if } p(x) \texttt{ then let } y = N; \texttt{true else true} \\
  &\cong \lambda^?x.\texttt{if } p(x) \texttt{ then true else true} \\
  &\cong \lambda^?x.\texttt{let } y = p(x); \texttt{if } y \texttt{ then true else true} \\
  &\cong \lambda^?x.\texttt{let } y = p(x); \texttt{true} \\
  &\cong \lambda^?x.\texttt{true}
  \end{aligned}$$

  Then by our parametricity assumption,

  $$Q_\forall[N]\ [?]\ V_t \cong \texttt{let } y = V_t; \texttt{true} \cong \texttt{true}$$

- Assume $\texttt{let } y = N; \texttt{true} \cong \mho$. We need to show that $Q_\forall[N]\ [?]\ V_t \cong \texttt{false}$.

  First, by β reductions and our assumption we know

  $$\begin{aligned}
  Q_?[N]\ V_t &= (\lambda^?x.\texttt{if } p(x) \texttt{ then let } y = N; \texttt{true else true})V_t \\
  &\cong \texttt{if } p(V_t) \texttt{ then let } y = N; \texttt{true else true} \\
  &\cong \texttt{if true then let } y = N; \texttt{true else true} \\
  &\cong \texttt{let } y = N; \texttt{true} \\
  &\cong \mho
  \end{aligned}$$

  and by similar reasoning $Q_?[N]\ V_f \cong \texttt{true}$.

  Next, by graduality we know that applying $Q_\forall[N]$ to the same inputs must approximate the behavior of $Q_?[N]$, so

  $$Q_\forall[N]\ [?]\ V_t \sqsubseteq Q_?[N]V_t \cong \mho$$

and

$$Q_\forall[N]\ [?]\ V_f \sqsubseteq Q_?[N]V_f \cong \texttt{true}$$

Then, by parametricity we also know that $Q_\forall[N]$ must be uniform, so

$$Q_\forall[N]\ [?]\ V_t \cong Q_\forall[N]\ [?]\ V_f$$

So we know that $Q_\forall[N]\ [?]\ V_t$ is below both $\mho$ and `true`, so by our ground value assumption we know

$$Q_\forall[?]\ V_t \cong \mho$$

which is our intended conclusion.

□

Then we can derive a contradiction by showing that this constructs a counter-example to Rice's theorem.

Theorem 187. *$\texttt{let}\ y = N; \texttt{true} \cong \texttt{true}$ is a decidable, non-trivial, semantic predicate on N.*

Proof.

- First, the predicate is semidecidable because $M \cong \texttt{true}$ is semidecidable for any M. Second, the predicate is co-semidecidable because $\texttt{let}\ y = N; \texttt{true} \ncong \texttt{true}$ if and only if $Q_\forall[N]\ [?]\ V_t \cong \mho$ which is semidecidable.
- Next, the predicate is non-trivial since $\texttt{let}\ y = \texttt{true}; \texttt{true} \cong \texttt{true}$ and $\texttt{let}\ y = \mho; \texttt{true} \cong \mho$ by one of our axioms.
- Finally, the predicate is semantic by congruence.

□

9.3 CONSEQUENCES

What conclusions should we draw from this? First, we should understand that the goal of correct dynamic enforcement of parametricity is unrealistic, and should be considered to be in the same category of properties as enforcing termination. But what is on the frontier of this impossibility results: which of our axioms can be violated in a reasonable system?

First, we might have a language in which there are *no* non-trivial total predicates on the dynamic type. This means introducing some error cases into things like tag tests. This is somewhat common in *dynamic sealing* approaches to enforcing parametricity, which we will examine in more detail in the next chapter. In these systems, the function $Q_\forall[N]$ above would always error, essentially because any use of a predicate p (think integer?) on a value of abstract type is considered

"non-parametric", whether or not the result of the predicate is used in a way that is extensionally non-parametric! So in particular the function

$$\lambda x.\texttt{if integer?}\, x\, \texttt{then true else true}$$

would be considered non-parametric, even though in a typical dynamically typed language it would be observationally equivalent to a constant function. From the perspective of a user of the original dynamic language, having this function be non-parametric would be considered *overly conservative*, punishing the function for utilizing integer? rather than the intended property of exhibiting non-parametric behavior. Furthermore, programmers may have used the assumption that a predicate was pure to justify refactorings, such as lifting a predicate into or out of a loop or callback, and so introducing these erroring behaviors might be considered a *backwards-incompatible* change to the language, and so should accordingly be carefully considered.

The next axiom we might drop is the one that says that it is semidecidable if $M \approx^{\text{ctx}} \mho$, i.e., to replace our finitary notion of an error with a more liberal notion of either erroring or diverging. This is more in line with the domain-theoretic notion of a universal domain than the gradual typing notion of a dynamic type. Such a system might be of interest for dynamically testing some functions to detect parametricity violations, but the fact that making types more precise might *introduce* divergence into a program means it is not very useful for a gradual typing system.

A final axiom we might drop is the ground value axiom that says that if a term M is inequivalent to a ground value like `true`, then the only behavior that can refine both terms is an error. This axiom is satisfied in all the logical relations models we have of the error ordering in this thesis, but all of these languages have a deterministic semantics. It may be possible to have a non-deterministic semantics where more non-deterministic terms are considered to be more precise, but we do not know of an existing model for this.

10

GRADUALITY AND PARAMETRICITY: TOGETHER AGAIN FOR THE FIRST TIME

Now we have seen that any approach to "enforcing" parametricity must adopt some compromise, we move to concrete language designs that attempt it anyway, using a mechanism called *dynamic sealing*. First, we will review some previous language designs, and why they all fail to satisfy at least one of graduality and parametricity. Then the remainder of the chapter will present an alternative language design that separates the gradual typing notion of type enforcement with the dynamic sealing mechanism altogether. The language using a different syntax from System F polymorphism, instead making alternative "fresh" universal and existential types that make dynamic sealing an explicit language feature rather than an opaque built-in enforcement scheme.

In previous chapters we have focused on how to make the graduality theorem true by ensuring that all casts arise from embedding-projection pairs, but now we review why graduality can fail. Languages can fail to satisfy the graduality theorem for a variety of reasons but a common culprit is *type-directed computation*. Whenever a form in a gradual language has behavior that is defined by inspection of the *type* of an argument, rather than by its *behavior*, there is a potential for a graduality violation, because the computation must be ensured to be *monotone* in the type. For instance, the Grace language supports a construct the designers call "structural type tests". That is, it includes a form M `is` A that checks if M has type A at runtime. Boyland [11] shows that care must be taken in designing the semantics of this construct if A is allowed to be an arbitrary type. For instance, it might seem reasonable to say that $(\lambda x : ?.x)$ `is Int` $\to$ `Int` should run to false because the function has type $? \to ?$. However, if we increase the precision of the types by changing the annotation, we get $(\lambda x : \texttt{Int}.x)$ `is Int` $\to$ `Int` which should clearly evaluate to true, violating the graduality principle. In such a system, we can't think of types as just properties whose precision can be tuned up or down: we also need to understand how changing the type might influence our use of type tests at runtime.

As discussed in earlier chapters (1, 2), graduality is crucial to a programmer easily reasoning about the migration process, and the more complicated the mechanisms of enforcement are, the more important it is to provide a system that provides graduality. Gradual typing researchers have designed languages that support reasoning principles enabled by a variety of advanced static features—such as ob-

jects [70, 79], refinement types [44], union and intersection types [12], typestates [89], effect tracking [66], subtyping [29], ownership [68], session types [41], and secure information flow [18, 23, 84]. As these typing features become more complicated, the behavior of casts can become sophisticated as well, and the graduality principle is a way of ensuring that these sophisticated mechanisms stay within programmer expectations.

Polymorphic languages can fail to satisfy the parametricity theorem for a variety of reasons but one common culprit is *type-directed computation* on abstract types. For instance in Java, values of a generic type T can be cast to an arbitrary object type. If the type T happens to be instantiated with the same type as the cast, then all information about the value will be revealed, and data abstraction is entirely lost. The problem is that the behavior of this runtime type-cast is directed by the type of the input: at runtime the input must carry some information indicating its type so that this cast can be performed. A similar problem arises when naïvely combining gradual typing with polymorphism, as we will see in §10.1.

While parametric polymorphism ensures data abstraction by means of a static type discipline, *dynamic sealing* provides a means of ensuring data abstraction even in a dynamically typed language. To protect abstract data from exposure, a fresh "key" is generated and implementation code must "seal" any abstract values before sending them to untrusted parties, "unsealing" them when they are passed into the exposed interface. For instance, we can ensure data abstraction for an untyped abstract functional counter by generating a fresh key σ, and producing a tuple where the first component is a 0 sealed with σ, and the increment and read-out function unseal their inputs and the increment function seals its output appropriately. If this is the only way the seal σ is used in the program, then the abstraction is ensured. While the programmer receives less support from the static type checker, this runtime sealing mechanism gives much of the same abstraction benefits.

One ongoing research area has been to satisfactorily combine the static typing discipline of parametric polymorphism with the runtime mechanism of dynamic sealing in a gradually typed language [4, 5, 42, 43, 85, 90]. However, no such language design has satisfied both of the desired fundamental theorems: graduality for gradual typing and relational parametricity for parametric polymorphism. Recent work by Toro, Labrada, and Tanter [85] claims to prove that graduality and parametricity are inherently incompatible, which backed by analogous difficulties for secure information flow [84] has led to the impression that the graduality property is incompatible with parametric reasoning. This would be the *wrong conclusion to draw*, for the following two reasons. First, the claimed proof has a narrow applicability. It is based on the definition of their logical relation, which we show in §10.1.3

does not capture a standard notion of parametricity. Second, and more significantly, we should be careful not to conclude that graduality and parametricity are incompatible properties, and that language designs must choose one. In this paper, we reframe the problem: both are desirable, and should be demanded of any gradual or parametric language. The failure of graduality and parametricity in previous work can be interpreted not as an indictment of these properties, but rather points us to reconsider the combination of System F's syntax with runtime semantics based on dynamic sealing. In this paper, we will show that graduality and parametricity are not in conflict *per se*, by showing that by modifying System F's syntax to make the sealing visible, both properties are achieved. Far from being in opposition to each other, both graduality and parametricity can be proven using a *single* logical relation theorem (§10.6).

10.1 GRADUALITY AND PARAMETRICITY, FRIENDS OR ENEMIES?

First, we review the issues in constructing a polymorphic gradual language that satisfies parametricity and graduality that have arisen in previous work. We see in each case that the common obstacle to parametricity and graduality is the presence of type-directed computation. This motivates our own language design, which obviates the need for type-directed computation by making dynamic sealing explicit in code.

10.1.1 *"Naïve" Attempt*

Before considering any dynamic sealing mechanisms, let's see why the most obvious combination of polymorphism with gradual typing produces a language that does not maintain data abstraction. Consider a polymorphic function of type $\forall X.X \to \texttt{Bool}$. In a language satisfying relational parametricity, we know that the function must treat its input as having abstract type X and so this input cannot have any influence on what value is returned. However, in a gradually typed language, any value can be *cast* using type ascriptions, such as in the function $\Lambda X.\lambda x : X.(x :: ?) :: \texttt{Bool}$. Here :: represents a type ascription. In a gradually typed language, a term M of type A can be ascribed a type B if it is "plausible" that an A is a B. This is typically formalized using a type consistency relation $\sim$ or more generally consistent subtyping relation $\lesssim$, but in either case, it is always plausible that an A is a ? and vice-versa, so in effect a value of any type can be cast to any other by taking a detour through the dynamic type. These ascriptions would then be elaborated to casts producing the term $\Lambda X.\lambda x : X.\langle \texttt{Bool} \Leftarrow ? \rangle \langle ? \Leftarrow X \rangle x$ If this function is applied to any value that is not compatible with `Bool`, then the function will error,

but if passed a boolean, the natural substitution-based semantics would result in the value being completely revealed:

$$(\Lambda X.\lambda x : X.\langle \texttt{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow X\rangle x)[\texttt{Bool}]\texttt{true} \mapsto^* \langle \texttt{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow \texttt{Bool}\rangle\texttt{true} \mapsto^* \texttt{true}$$

From a semantic perspective this should not be surprising. Polymorphic functions allow for the parameterization of a function over a *type*, but the notion of type differs between different languages. In Chapter 3, we made a model where a gradual type A was interpreted as a pair of a logical type $|A|$ and an ep pair $|A| \lhd ?$. This "naïve" semantics is just the natural result of this idea: polymorphic functions receive both the logical type and the corresponding ep pair as inputs, which allows them to implement casts such as $\langle \texttt{Bool} \Leftarrow ?\rangle$. This is the root-cause of this parametricity violation is that we allow casts like $\langle ? \Leftarrow X\rangle$ whose behavior depends on how X is instantiated. To construct a gradual language with strong data abstraction we must somehow avoid the dependency of $\langle ? \Leftarrow X\rangle$ on X.

One option, is to ban casts like $\langle ? \Leftarrow X\rangle$ altogether. Syntactically, this means changing the notion of plausibility to say that ascribing a term of type X with the dynamic type ? is not allowed. Semantically, this corresponds to only quantifying over the logical type $|X|$, and not having access to the ep pair. This is possible using the system presented by Igarashi, Sekiyama, and Igarashi [42] if you only allow Λs that use the "static" label. This is compatible with parametricity and graduality, but is somewhat against the spirit of gradual typing, where typically all programs could be written as dynamically typed programs, and dynamically typed functions can be used on values of any type.

A fairly ingenious alternative, based on the insights of Morris [52], is called *dynamic sealing*. The basic idea is that polymorphic functions will still be parameterized by a logical type and an ep pair, but we ensure that the ep pair is always constructed by a fresh tag on the dynamic type, So dynamic sealing allows casts like $\langle ? \Leftarrow X\rangle$, but ensures that their behavior does not depend on *how X is instantiated*.

10.1.2 *Type-directed Sealing*

In sealing-based gradual parametric languages like λB [4, 5], we ensure that casts of abstract type do not depend on their instantiation by adding a layer of indirection. Instead of the usual β rule for polymorphic functions

$$(\Lambda X.M)[A] \mapsto M[A/X],$$

in λB, we dynamically generate a fresh type α and pass that in for X. This first of all means the runtime state must include a store of fresh

types, written Σ. When reducing a type application, we generate a fresh type α and instantiate the function with this new type

$$\Sigma; (\Lambda X.M)[A] \mapsto \Sigma, \alpha := A; M[\alpha/X]$$

In this case, we interpret α as being a new tag on the dynamic type that tags values of type A but is different from all previously used tags. The casts involving α are treated like a new base type, incompatible with all existing types. However, if we look at the resulting term, it is not well-typed: if the polymorphic function has type $\forall X.B$, then $M[\alpha/X]$ has type $B[\alpha/X]$, but the context of this term expects it to be of type $B[A/X]$. To paper over this difference, λB wraps the substitution with a *type-directed coercion*, distinct from casts, that mediates between the two types:

$$\Sigma; (\Lambda X.M)[A] \mapsto \Sigma, \alpha := A; M[\alpha/X] : B[\alpha/X] \stackrel{+\alpha}{\Longrightarrow} B[A/X]$$

This type-directed coercion $M[\alpha/X] : B[\alpha/X] \stackrel{+\alpha}{\Longrightarrow} B[A/X]$ is the part of the system that performs the actual sealing and unsealing, and is defined by recursion on the type B. The $+\alpha$ indicates that we are *unsealing* values in positive positions and sealing at negative positions. For instance if $B = X \times \texttt{Bool}$, and $X = \texttt{Bool}$, then on a pair $(\texttt{seal}_\alpha\texttt{true}, \texttt{false})$ the coercion will unseal the sealed boolean on the left and leave the boolean on the right alone. If B is of function type, the definition will involve the dual coercion using $-\alpha$, which *seals* at positive positions. So for instance applying the polymorphic identity function will reduce as follows

$$\begin{aligned}
&\Sigma; (\Lambda X.\lambda x : X.x)[\texttt{Bool}]\texttt{true} \\
&\mapsto \Sigma, \alpha := \texttt{Bool}; (\lambda x : \alpha.x : \alpha \to \alpha \stackrel{+\alpha}{\Longrightarrow} \texttt{Bool} \to \texttt{Bool})\texttt{true} \\
&\mapsto \Sigma, \alpha := \texttt{Bool}; (\lambda x : \alpha.x)(\texttt{true} : X \stackrel{-\alpha}{\Longrightarrow} \alpha) : \alpha \stackrel{+\alpha}{\Longrightarrow} X \\
&\mapsto \Sigma, \alpha := \texttt{Bool}; (\lambda x : \alpha.x)(\texttt{seal}_\alpha\texttt{true}) : \alpha \stackrel{+\alpha}{\Longrightarrow} X \\
&\mapsto \Sigma, \alpha := \texttt{Bool}; \texttt{seal}_\alpha\texttt{true} : \alpha \stackrel{+\alpha}{\Longrightarrow} X \\
&\mapsto \texttt{true}
\end{aligned}$$

While this achieves the goal of maintaining data abstraction, it unfortunately violates graduality, as first pointed out by Igarashi, Sekiyama, and Igarashi [42]. The reason is that the coercion is a type-directed computation, this time directed by the type $\forall X.B$ of the polymorphic function, whose behavior observably differs at type X from its behavior at type ?. Specifically, a coercion $M : X \stackrel{-\alpha}{\Longrightarrow} \alpha$ results in sealing the result of M, whereas if X is replaced by dynamic, then $M : ? \stackrel{-\alpha}{\Longrightarrow} \alpha$ is an identity function. An explicit counter-example is given by modifying the identity function to include an explicit annotation. The term $M_1 = (\Lambda X.\lambda x : X.x :: X)[\texttt{Bool}]\texttt{true}$ reduces by generating a seal α, sealing the input `true` with α, then unsealing it, finally producing `true`. On the other hand, if the type of the input were dynamic rather

than X, we would get a term $M_2 = (\Lambda X.\lambda x : ?.(x :: X))[\texttt{Bool}]\texttt{true}$. In this case, the input is *not* sealed by the implementation, and the ascription of X results in a failed cast since `Bool` is incompatible with α. The only difference between the two terms is a type annotation, meaning that $M_1 \sqsubseteq M_2$ in the term precision ordering (M_1 is more precise than M_2), and so the graduality theorem states that if M_1 does not error, it should behave the same as M_2, but in this case M_2 errors while M_1 does not. The problem here is that the type of the polymorphic function determines whether to seal or unseal the inputs and outputs, but graduality says that the behavior of the dynamic type must align with both abstract types X (indicating sealing/unsealing) and concrete types like `Bool` (indicating no sealing/unsealing). These demands are contradictory since dynamic code would have to simultaneously be opaque until unsealing and available to interact with. So we see that the attempt to remove the type-directed casts which break parametricity by using dynamic sealing led to the need for a type-directed coercion which breaks graduality.

10.1.3 *To Seal, or not to Seal*

The language GSF was introduced by Toro, Labrada, and Tanter [85] to address several criticisms of the type system and semantics of λB. We agree with the criticisms of the type system and so we will focus on the semantic differences. GSF by design has the same violation of graduality as λB, but has different behavior when using casts.

One motivating example for GSF is what happens when casting the polymorphic identity function to have a dynamically typed output: $(((\Lambda X.\lambda x : X.x) :: \forall X.X \to ?)[\texttt{Int}]1) + 2$. In λB, the input 1 is sealed as dictated by the type, but the dynamically typed output is not unsealed when it is returned from the function, resulting in an error when we try to add it. Ahmed et al. [4] argue that it should be a free theorem that the behavior of a function of type $\forall X.X \to ?$ should be independent of its argument: it always errors, diverges or it always returns the same dynamic value, based on the intuition that the dynamic type ? does not syntactically contain the free variable X, and that this free theorem holds in System F. This reasoning is suspect since at runtime, the dynamic type does include a case for the freshly allocated type X, so intuitively we should consider ? to include X (and any other abstract types in scope).

Toro, Labrada, and Tanter [85] argue on the other hand that intuitively the identity function was written with the intention of having a sealed input that is returned and then unsealed, and so casting the program to be more dynamic should result in the same behavior and so the program should succeed. The function application runs to the equivalent of $\langle ? \Leftarrow \texttt{Int} \rangle 1$ which is then cast to `Int` and added to 2, resulting in the number 3. The mechanism for achieving this semantics

is a system of runtime *evidence*, based on the *Abstracting Gradual Typing* (AGT) framework [29]. An intuition for the behavior is that the sealing is still type-directed, but rather than being directed by the *static* type of the function being instantiated, it is based on the most precise type that the function has had. So here because the function was originally of type $\forall X.X \to X$, the sealing behavior is given by that type.

However, while we agree that the analysis in Ahmed et al. [4] is incomplete, the behavior in GSF is inherently non-parametric, because the polymorphic program produces values with *different* dynamic type tags based on what the input *type* is. As a user of this function, we should be able to replace the instantiating type `Int` with `Bool` and give any boolean input and get related behavior at the type ?, but in the program $(((\Lambda X.\lambda x : X.x) :: \forall X.X \to ?)[\texttt{Bool}]\texttt{true}) + 2$ the function application reduces to $\langle ? \Leftarrow \texttt{Bool}\rangle\texttt{true}$ which errors when cast to `Int`. Intuitively, this behavior is not parametric because the first program places an `Int` tag on its input, and the second places a `Bool` tag on its input.

The non-parametricity is clearer if we look at a program of type $\forall X.? \to \texttt{Bool}$ and consider the following function, a constant function with abstract input type cast to have dynamic input:

$$\text{const} = (\Lambda X.\lambda x : X.\texttt{true}) :: \forall X.? \to \texttt{Bool}$$

X now has no effect on static typing, so both const$[\texttt{Int}]3$ and const$[\texttt{Bool}]$ are well-typed. However, since the sealing behavior is actually determined by the type $\forall X.X \to \texttt{Bool}$, the program will try to seal its input after downcasting it to whatever type X is instantiated at. So the first program casts $\langle \texttt{Int} \Leftarrow ?\rangle\langle ? \Leftarrow \texttt{Int}\rangle 3$, which succeeds and returns `true`, while the second program performs the cast $\langle \texttt{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow \texttt{Int}\rangle 3$ which fails. In effect, we have implemented a polymorphic function that for any type X, is a recognizer of dynamically typed values for that type, returning `true` if the input matches X and erroring otherwise. Any implementation of this behavior would clearly require passing of some syntactic representation of types at runtime.

Formally, the GSF language does not satisfy the following *defining principle of relational parametricity*, as found in standard axiomatizations of parametricity such as Dunphy [19], Ma and Reynolds [48], and Plotkin and Abadi [63]. In a parametric language, the user of a term M of a polymorphic function type $\forall X.A \to B$ should be guaranteed that M will behave uniformly when instantiated multiple times. Specifically, a programmer should be able to instantiate M with two different types B_1, B_2 and choose any relation $R \in Rel[B_1, B_2]$ (where the notion of relation depends on the type of effects present), and be ensured that if they supply related inputs to the functions, they will get related

outputs. Formally, for a Kripke-style relation, the following principle should hold:

$$\frac{M : \forall X.A \to B \qquad R \in \mathrm{Rel}[B_1, B_2] \qquad (w, V_1, V_2) \in \mathcal{V}[\![A]\!]\rho[X \mapsto R]}{(w, M[B_1]V_1, M[B_2]V_2) \in \mathcal{E}[\![B]\!]\rho[X \mapsto R]}$$

Here w is a "world" that gives the invariants in the store and ρ is the relational interpretation of free variables. $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{E}[\![\cdot]\!]$ are value and expression relations formalizing an approximation ordering on values and expressions respectively, and $X \mapsto R$ means that the relational interpretation of X is given by R.

Toro, Labrada, and Tanter [85] use an unusual logical relation for their language based on a similar relation in Ahmed et al. [5], so there is no direct analogue of the relational mapping $X \mapsto R$. Instead, the application extends the world with the association of α to R and the interpretation sends X to α. However, we can show that this parametricity principle is violated by *any* ρ we pick for the term const above, using the definition of $\mathcal{E}[\![\cdot]\!]$ given in [85][1]. Instantiating the lemma would give us that $(w, \mathrm{const}[\texttt{Int}]3, \mathrm{const}[\texttt{Bool}]3) \in \mathcal{E}[\![\texttt{Bool}]\!]\rho$ since $(w, 3, 3) \in \mathcal{V}[\![?]\!]\rho$ for any ρ. The definition of $\mathcal{E}[\![\texttt{Bool}]\!]\rho$ then says (again for any ρ) that it should be the case that since $\mathrm{const}[\texttt{Int}]3$ runs to a value, it should also be the case that $\mathrm{const}[\texttt{Bool}]3$ runs to a value as well, but in actuality it errors, and so this parametricity principle must be false.

How can the above parametricity principle be false when Toro, Labrada, and Tanter [85] prove a parametricity theorem for GSF? We have not found a flaw in their *proof*, but rather a mismatch between their *theorem* statement and the expected meaning of parametricity. The definition of $\mathcal{V}[\![\forall X.A]\!]$ in Toro, Labrada, and Tanter [85] is not the usual interpretation, but rather is an adaptation of a non-standard definition used in Ahmed et al. [5]. Neither of their definitions imply the above principle, so we argue that neither paper provides a satisfying proof of parametricity. With GSF, we see that the above behavior violates some expected parametric reasoning, using the definition of $\mathcal{V}[\![?]\!]$ given in Toro, Labrada, and Tanter [85]. With λB, we know of no counterexample to the above principle, and we conjecture that it would satisfy a more standard formulation of parametricity.

It is worth noting that the presence of effects—such as nontermination, mutable state, control effects—requires different formulations of the logical relation that defines parametricity. However, those logical relations capture parametricity in that they always formalize *uniformity of behavior* across different type instantiations. For instance, for a language that supports nontermination, the logical relation for parametricity ensures that two different instantiations have the same termination behavior: either both diverge, or they both terminate with

1 they use slightly different notation, but we use notation that matches the logical relation we present later

related values. Because of this, the presence of effects usually leads to weaker free theorems—in pure System F all inhabitants of $\forall X.X \to X$ are equivalent to the identity function, but in System F with non-termination, every inhabitant is either the identity or *always* errors. Though the free theorems are weaker, parametricity still ensures *uniformity* of behavior. As our counterexample above (const[`Int`]3 vs. const[`Bool`]3) illustrates, GSF is non-parametric since it does not ensure uniform behavior. However, since the difference in behavior was between error and termination, it is possible that GSF satisfies a property that could be called "partial parametricity" (or parametricity modulo errors) that weakens the notion of uniformity of behavior: either one side errors or we get related behaviors. However, it is not clear to us how to formulate the logical relation for the dynamic type to prove this. We show how this weakened reasoning in the presence of ? compares to reasoning in our language PolyG$^\nu$ in §10.7.

Our counter-example crucially uses the dynamic type, and we conjecture that when the dynamic type does not appear under a quantifier, that the usual parametric reasoning should hold in GSF. This would mean that in GSF once polymorphic functions become "fully static", they support parametric reasoning, but we argue that it should be the goal of gradual typing to support type-based reasoning even in the presence of dynamic typing, since migration from dynamic to static is a gradual process, possibly taking a long time or never being fully completed.

10.1.4 *Resolution: Explicit Sealing*

Summarizing the above examples, we see that

1. The naïve semantics leads to type-directed casts at abstract types, violating parametricity.

2. λB's type-directed sealing violates graduality because of the ambiguity of whether or not the dynamic type indicates sealing/unsealing or not.

3. GSF's variant of type-directed sealing based on the most precise type violates graduality as the others do, but also violates parametricity because the polymorphic function gets to determine which dynamically typed values are sealed (i.e. abstract) and which are not.

We see that in each case, the use of a type-directed computational step leads to a violation of graduality or parametricity. The GSF semantics makes the type-directed sealing of λB more flexible by using the runtime evidence attached to the polymorphic function rather than the type at the instantiation point, but unfortunately this makes it

impossible for the continuation to reason about which dynamically typed values it passes will be treated as abstract or concrete.

In the remainder of this chapter we will present the syntax and semantics of PolyG$^\nu$, an alternative approach to incorporating dynamic sealing with gradual typing that avoids the need for any type-directed sealing. We designed PolyG$^\nu$by revisiting some of the basic ideas of dynamic sealing approaches. First, notice that the dynamic "enforcement" of parametricity provided by dynamic sealing approaches is fundamentally different from the gradual typing semantics of previous chapters in that it takes place regardless of whether or not any casts occur in the program. Whenever we instantiate a polymorphic type with a variable, a new type tag is created and the inputs and outputs are sealed and unsealed as dictated by the types, even if the only type used is the dynamic type. The sealing affects what certain casts *mean*, such as casting from abstract X to ? or vice-versa, but the sealing itself does not arise because some not necessarily parametric function is cast to parametric type. If dynamic sealing is "enforcing" parametricity, it is "enforcing" it on all polymorphic functions, even those that are completely written in the statically typed portion of the language.

As we saw in the previous chapter, the idea of "enforcing" parametricity is fundamentally flawed, and so in PolyG$^\nu$ we abandon the idea entirely. Instead, we think of the dynamic sealing version of universal and existential types as being a *new type*, fundamentally different from the System F notion of universal and existential types, but similar enough to provide some of the same benefits.

The dynamic sealing version of universally quantified types, which we will call *fresh universal* types, ensures that any instantiation of the polymorphic function uses a *fresh* type. However recall that when we use the syntax of System F instantiation:

$$\frac{M : \forall X.A}{M[B] : A[B/X]}$$

We get a mismatch, instead of instantiating with B we instantiate with a fresh type α that is merely isomorphic to B, not the same as it.

$$\Sigma; (\Lambda X.M)[A] \mapsto \Sigma, \alpha := A; M[\alpha/X]$$

This results in the need for type-directed sealing to mask the difference. This is the main difference in PolyG$^\nu$. In PolyG$^\nu$, we make manifest the fact that when we instantiate a fresh polymorphic function, it creates a new type. Then, the continuation of the instantiation must use this fresh type, rather than the original. To give a simple scoping for this, we need to introduce an ANF-restriction to instantiations, giving us the syntax

$$\texttt{let}\ x = M\{X \cong B\}; N$$

Here M is instantiated with the type X, which is a *newly generated* type that is constructed to be isomorphic to, but not identical to, B. The result is then bound to x and the continuation N now has an explicit isomorphism in scope $\texttt{seal}_X : B \to X$ and $\texttt{unseal}_X : X \to B$ allows it to manually seal and unseal inputs and outputs of x as necessary. Then rather than the runtime system implementing the complex system of sealing by induction over types, the programmer explicitly seals in the program, and so avoids the need for type-directed sealing.

A similar syntax works for existential types, but the instantiating party is the *introduction* form

$$\texttt{pack}^{\nu}(X \cong A, M)$$

Which when unpacked will create a fresh type X isomorphic to A in M. We can think of this as a simplistic kind of module system where X is a new type created to be isomorphic to A and is exported without exposing this isomorphism, resembling the use of modules in Haskell to provide data abstraction.

Summarizing the high level ideas behind PolyG$^{\nu}$:

1. We depart from the syntax of System F.
2. Sealing/unsealing of values is explicit and programmable, rather than implicit and type-directed.
3. The party that *instantiates* an abstract type is the party that determines which values are sealed and unsealed. For existential types, this is the package (i.e., the module) and dually for universal types it is the continuation of the instantiation.

The dynamic semantics of PolyG$^{\nu}$ are similar to λB without the type-directed coercions, removing the obstacle to proving the graduality theorem. By allowing user-programmable sealing and unsealing, more complicated forms of sealing and unsealing are possible: for instance, we can seal every prime number element of a list, which would require a very rich type system to express using type-directed sealing! We conjecture that the language is strictly more expressive than λB in the sense of Felleisen [22]: λB should be translatable into PolyG$^{\nu}$ in a way that simulates its operational semantics. Because the sealing is performed by the instantiating party rather than the abstracting party, the expressivity of PolyG$^{\nu}$ is incomparable to GSF.

We summarize the contributions of this chapter as follows

- We identify type-directed computation as the common cause of graduality and parametricity violations in previous work on gradual polymorphism.
- We show that certain polymorphic programs in Toro, Labrada, and Tanter [85]'s language GSF exhibit non-parametric behavior.
- We present a new surface language PolyG$^\nu$ that supports a novel form of universal and existential types where the creation of fresh types is exposed in a controlled way. The semantics of PolyG$^\nu$ is similar to previous gradual parametric languages, but the explicit type creation and sealing eliminates the need for type-directed computation.
- We elaborate PolyG$^\nu$ into an explicit cast calculus PolyC$^\nu$. We then give a translation from PolyC$^\nu$ into a typed target language, CBPV$_{\text{OSum}}$, essentially call-by-push-value with polymorphism and an extensible sum type.
- We develop a novel logical relation that proves both graduality and parametricity for PolyG$^\nu$. Thus, we show that parametricity and graduality are compatible, and we strengthen the connection alluded to by New and Ahmed [56] that graduality and parametricity are analogous properties.

10.2 PolyG$^\nu$: A GRADUAL LANGUAGE WITH POLYMORPHISM AND SEALING

Next, we present a our gradual language, PolyG$^\nu$, that supports a variant of existential and universal quantification while satisfying parametricity and graduality. The language has some unusual features, so we start with an extended example to illustrate what programs look like, and then in §10.2.2 introduce the formal syntax and typing rules.

10.2.1 *PolyG$^\nu$ Informally*

Let's first consider an example of existential types, since they are simpler than universal types in PolyG$^\nu$. In a typed, non-gradual language, we can define an abstract "flipper" type, $\texttt{FLIP} = \exists X.X \times (X \to X) \times (X \to \texttt{Bool})$. The first element is the initial state, the second is a "toggle" function and the last element reads out the value as a concrete boolean.

Then we could create an instance of this abstract flipper using booleans as the carrier type X and negation as the toggle function $\texttt{pack}(\texttt{Bool}, (\texttt{true}, (\texttt{NOT}, \texttt{ID})))\ \texttt{as}\ \texttt{FLIP}$. Note that we must explicitly

mark the existential package with a type annotation, because otherwise we wouldn't be able to tell which occurrences of `Bool` should be hidden and which should be exposed. With different type annotations, the same package could be given types $\exists X.\texttt{Bool} \times (\texttt{Bool} \to \texttt{Bool}) \times (\texttt{Bool} \to \texttt{Bool})$ or $\exists X.X \times (X \to X) \times (X \to X)$.

The PolyG$^\nu$ language existential type works differently in a few ways. We write $\exists^\nu$ rather than $\exists$ to emphasize that we are only quantifying over fresh types, and not arbitrary types. The equivalent of the above existential package would be written as

$$\texttt{pack}^\nu(X \cong \texttt{Bool}, (\texttt{seal}_X\texttt{true}, ((\lambda x : X.\texttt{seal}_X(\texttt{NOT}(\texttt{unseal}_X x))), (\lambda x : X.\texttt{unseal}_X x)))$$

The first thing to notice is that rather than just providing a type `Bool` to instantiate the existential, we write a declaration $X \cong \texttt{Bool}$. The X here is a *binding position* and the body of the package is typed under the assumption that $X \cong \texttt{Bool}$. Then, rather than *substituting* `Bool` for X when typing the body of the package, the type checker checks that the body has type $X \times ((X \to X) \times (X \to \texttt{Bool}))$ under the assumption that $X \cong \texttt{Bool}$: Crucially, $X \cong \texttt{Bool}$ is a *weaker* assumption than $X = \texttt{Bool}$. In particular, there are no *implicit* casts from X to `Bool` or vice-versa, but the programmer can *explicitly* "seal" `Bool` values to be X using the form $\texttt{seal}_X M$, which is only well-typed under the assumption that $X \cong A$ for some A consistent with `Bool`. We also get a corresponding unseal form $\texttt{unseal}_X M$, and the runtime semantics in §10.4.4 defines these to be a bijection. At runtime, X will be a freshly generated type with its own tag on the dynamic type. An interesting side-effect of making the difference between X and `Bool` explicit in the term is that existential packages do not require type annotations to resolve any ambiguities. For instance, unlike in the typed case, the gradual package above could *not* be ascribed the type $\exists^\nu X.\texttt{Bool} \times ((\texttt{Bool} \to \texttt{Bool}) \times (\texttt{Bool} \to \texttt{Bool}))$ because the functions explicitly take X values, and not `Bool` values.

The corresponding elimination form for $\exists^\nu$ is a standard unpack: $\texttt{unpack}\ (X, x) = M; N$, where the continuation for the unpack is typed with just X and x added to the context, it doesn't know that $X \cong A$ for any particular A. We call this ordinary type variable assumption an *abstract type variable*, whereas the new assumption $X \cong A$ is a *known type variable* which acts more like a *type definition* than an abstract type. At runtime, when an existential is unpacked, a fresh type X is created that is isomorphic to A but whose behavior with respect to casts is different.

While explicit sealing and unsealing might seem burdensome to the programmer, note that this is directly analogous to a common pattern in Haskell, where modules are used in combination with `newtype` to create a datatype that at runtime is represented in the same way as an existing type, but for type-checking purposes is considered distinct. We give an analogous Haskell module as follows:

```
module Flipper(State, start, toggle, toBool) where

newtype State = Seal { unseal :: Bool }

start :: State
start = Seal True

inc :: State -> State
inc s = Seal (not (unseal s)

toBool :: State -> Bool
toBool = unseal
```

Then a different module that imports `Flipper` is analogous to an unpack, as its only interface to the `State` type is through the functions provided.

We also add universal quantification to the language, using the duality between universals and existentials as a guide. Again we write the type differently, as $\forall^{\nu} X.A$. In an ordinary polymorphic language, we would write the type of the identity function as $\forall X.X \to X$ and implement it using a Λ form: $\Lambda X.\lambda x : X.x$. The elimination form passes in a type for X. For instance applying the identity function to a boolean would be written as $\texttt{ID}\,[\texttt{Bool}]\ \texttt{true}$. And a free theorem tells us that this term must ultimately diverge, error, or return `true`.

The introduction form Λ^{ν} is dual to the unpack form, and correspondingly looks the same as the ordinary Λ, for example in the identity function

$$\texttt{ID}^{\nu} = (\Lambda^{\nu} X.\lambda x : X.x) : \forall^{\nu} X.X \to X$$

The body of the Λ^{ν} is typed with an abstract type variable X in scope. The elimination form of type application is dual to the *pack* form, and so similarly introduces a *known* type variable assumption. Instantiating the identity function as above would be written as

$$(\texttt{let}\ f = \texttt{ID}^{\nu}\{X \cong \texttt{Bool}\}; \texttt{unseal}_X(f\ (\texttt{seal}_X \texttt{true}))) : \texttt{Bool}$$

which introduces a known type variable $X \cong \texttt{Bool}$ into the context. Rather than the resulting type being $\texttt{Bool} \to \texttt{Bool}$, it is $X \to X$ with the assumption $X \cong \texttt{Bool}$. Then the argument to the function must be explicitly sealed as an X to be passed to the function. The output of the function is also of type X and so must be explicitly *unsealed* to get a boolean out. However since the $X \cong \texttt{Bool}$ needs to be bound in the *continuation* of the application, we introduce an explicit binding form, giving us a somewhat inconvenient ANF-like restriction on universal types. In the original paper version of this Chapter, we consider an alternative "inside-out" binding structure, but present the simpler ANF restriction here. While unusual, the elimination form is intuitively justified by the duality with existentials: we can think of

the *continuation* for an instantiation of a $\forall^\nu$ as being analogous to the *body* of the existential package.

To get an understanding of how PolyG$^\nu$ compares to λB and GSF and why it avoids their violation of graduality, let's consider how we might write the examples from the previous section. In PolyG$^\nu$, if we apply a function of type $\forall X.X \to X$, we have to mark explicitly that the input is sealed, and furthermore if we want to use the output as a boolean, we must *unseal* the output:

$$\texttt{let } f = (\Lambda X.\lambda x : X.x :: X)\{X \cong \texttt{Bool}\}; (\texttt{unseal}_X(f\ (\texttt{seal}_X\texttt{true}))) \mapsto^* \texttt{true}$$

Then if we change the type of the input from X to ? the explicit sealing and unsealing remain, so even though the input is dynamically typed it will still be a sealed boolean, and the program exhibits the same behavior:

$$\texttt{let } f = (\Lambda X.\lambda x : ?.x :: X)\{X \cong \texttt{Bool}\}; (\texttt{unseal}_X(f\ (\texttt{seal}_X\texttt{true}))) \mapsto^* \texttt{true}$$

If we remove the seal of the input, then the cast to X in the function will fail, giving us the behavior of λB/GSF:

$$\texttt{let } f = (\Lambda X.\lambda x : ?.x :: X)\{X \cong \texttt{Bool}\}; (\texttt{unseal}_X(f\ \texttt{true})) \mapsto^* \mathcal{U}$$

but there are two crucial differences with λB/GSF. First, this involved changing the term, not just the type, so the graduality theorem does not tell us that the programs should have related behavior. Second, we don't think of this error that we get as a "parametricity violation" error, but simply the entirely normal "tag mismatch" dynamic type error, where we explicitly tagged certain values as being of the new type X in one program and left it as `Bool`-tagged in the other.

Next, let's consider the parametricity violation from GSF. When we instantiate the constant function, the programmer, not the runtime system, decides if the argument is sealed or not. We get the behavior of GSF when we instantiate with `Int` and seal the input 3:

$$\texttt{let } f = \text{const}\{X \cong \texttt{Int}\}; f\ (\texttt{seal}_X 3) \mapsto^* \texttt{true}$$

However, if we try to write the analogous program with `Bool`: instead of `Int`

$$\texttt{let } f = \text{const}\{X \cong \texttt{Bool}\}; f\ (\texttt{seal}_X 3)$$

then the program is not well typed because $X \cong \texttt{Bool}$ and 3 has type `Int` which is not compatible. We can replicate the outcome of the GSF program by *not* sealing the 3:

$$\texttt{let } f = \text{const}\{X \cong \texttt{Bool}\}; f\ 3$$

But this is not a parametricity violation because the 3 here will be embedded at the dynamic type with the `Int` tag, whereas above the 3 was tagged with the X tag, which is not related.

$$
\begin{array}{llll}
\text{types} & A, B & ::= & ? \mid X \mid \texttt{Bool} \mid A \times B \mid A \to B \mid \exists^{\nu} X.A \mid \forall^{\nu} X.A \\
\text{Ground types} & G & ::= & X \mid \texttt{Bool} \mid ? \times ? \mid ? \to ? \mid \exists^{\nu} X.? \mid \forall^{\nu} X.? \\
\text{terms} & M, N & ::= & x \mid M :: A \mid \texttt{seal}_X M \mid \texttt{unseal}_X M \mid \texttt{is}(G)\texttt{?}\ M \\
 & & \mid & \texttt{let}\ x = M; N \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \\
 & & \mid & (M, N) \mid \texttt{let}\ (x, y)\ =\ M; N \\
 & & \mid & \lambda x : A.M \mid M\ N \\
 & & \mid & \texttt{pack}^{\nu}(X \cong A, M) \mid \texttt{unpack}\ (X, x) = M; N \\
 & & \mid & \Lambda^{\nu} X.M \mid \texttt{let}\ x = M\{X \cong A\}; N \\
\text{environment} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, X \mid \Gamma, X \cong A
\end{array}
$$

Figure 10.1: PolyG$^{\nu}$ Syntax

$$
\vdash \cdot \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, X} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, X \cong A}
$$

$$
\Gamma \vdash ? \qquad \frac{X \in \Gamma}{\Gamma \vdash X} \qquad \Gamma \vdash \texttt{Bool} \qquad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}
$$

$$
\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \frac{\Gamma, X \vdash A}{\Gamma \vdash \exists^{\nu} X.A} \qquad \frac{\Gamma, X \vdash A}{\Gamma \vdash \forall^{\nu} X.A}
$$

Figure 10.2: Well-formedness of Environments, Types

10.2.2 *PolyG$^{\nu}$ Formal Syntax and Semantics*

Figure 10.1 presents the syntax of the surface language types, terms and environments. Most of the language is a typical gradual functional language, using ? as the dynamic type, and including type ascription $M :: A$. The unusual aspects of the language are the $\texttt{seal}_X M$ and $\texttt{unseal}_X M$ forms and the "fresh" existential $\exists^{\nu} X.A$ and universal $\forall^{\nu} X.A$. Note also the non-standard environments Γ, which include ordinary typing assumptions $x : A$, abstract type variable assumptions X and known type variable assumptions $X \cong A$. For simplicity, we assume freshness of all type variable bindings, i.e. when we write Γ, X or $\Gamma, X \cong A$ that X does not occur in Γ. The well-formedness of these environments is mutually defined with well-kindedness of types in Figure 10.2

The typing rules are presented in Figure 10.3. We follow the usual formulation of gradual surface languages in the style of Siek and Taha [69]: type checking is strict when checking compatibility of different connectives, but lax when the dynamic type is involved. The first rule GAnn for the type annotation form $M :: B$ says the term is well

formed when the type of the term A is *consistent* with the annotation B, codified in the *type consistency* relation $A \sim B$. We define this in the standard way below the typing rules as being the least congruence relation including equality and rules making ? consistent with every type.

We include variable (GVar) and let-binding (GLet) rules. Next, the sealing rule (GSeal) says we can seal a term M's value at a know type variable X if the type A that X is known to be isomorphic to ($X \cong A \in \Gamma$) is consistent with the type of M. The unseal rule (GUnseal) gives access to the opposite direction of the isomorphism, requiring that the type of M be consistent with the given known type variable X. It is crucial for graduality to hold that this bijection is explicit and not implicit, because the behavior of casts involving X and A are very different. To show that PolyG$^\nu$ is compatible with total predicates, we also include a form `is(`G`)?` M that checks at runtime whether M returns a value that has the tag associated with the ground type G. The rule (GCheckTag) states that M can have any type in this case because it is always a safe operation, but the result is either trivially `true` or `false` unless M has type ?. For example, if M has static type X then `is(`X`)?` M will always evaluate to `true` (if M reduces to a value) and if M has any other non-dynamic type, such as $A \to B$, then this will always evaluate to `false`.

Next, we have boolean values (GTrue, GFalse) `true` and `false`. Then the GIf rule for an if-statement checks that the scrutinee has a type compatible with `Bool`, and as in previous work uses gradual meet $B_t \sqcap B_f$ for the output type [28]. Gradual meet, defined in the bottom right of the figure, is a partial operation, since this ensures that if the two sides have different (non-?) head connectives then type checking errors.

Next, we have pairs (GPair,GPmPair) and functions (GFun,GApp), which are fairly standard. We use pattern-matching as the elimination form for pairs. To reduce the number of rules, we present the elimination forms in the style of Garcia and Cimini [28], using partial functions $\pi_i, \mathrm{dom}, \mathrm{cod}$ and later $\mathrm{un}\forall^\nu, \mathrm{un}\exists^\nu$ to extract the subformula from a type "up to ?". For the correct type this extracts the actual subformula, but for ? is defined to be ? and for other connectives is undefined. We define these at the bottom left of the figure, where uncovered cases are undefined. Next, we have existentials, which are as described in §10.2.1. The introduction rule (GPack) introduces a known type variable $X \cong A$ when type checking the body of the pack, rather than substituting A for X as in the System F-style rule. In the elimination rule (GUnpack), we essentially destructure the scrutinee to get out an unknown type variable X and the value of type $\mathrm{un}\exists^\nu{}_X(A)$ where A is the type of the scrutinee M. Note that unlike the other partial operations, $\mathrm{un}\exists^\nu{}_X$ (and $\mathrm{un}\forall^\nu{}_X$) is indexed by a type variable since it potentially is instantiating a bound variable. Similar

$$\textsc{GAnn}\ \frac{\Gamma \vdash M : A \qquad A \sim B}{\Gamma \vdash (M :: B) : B} \qquad \textsc{GVar}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{GLet}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \texttt{let}\ x = M; N : B}$$

$$\textsc{GSeal}\ \frac{\Gamma \vdash M : B \qquad X \cong A \in \Gamma \qquad B \sim A}{\Gamma \vdash \texttt{seal}_X M : X} \qquad \textsc{GUnseal}\ \frac{\Gamma \vdash M : B \qquad X \cong A \in \Gamma \qquad B \sim X}{\Gamma \vdash \texttt{unseal}_X M : A} \qquad \textsc{GCheckTag}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash G}{\Gamma \vdash \texttt{is}(G)\texttt{?}\ M : \texttt{Bool}}$$

$$\textsc{GTrue}\ \Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \textsc{GFalse}\ \Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \textsc{GIf}\ \frac{\Gamma \vdash M : A \qquad A \sim \texttt{Bool} \qquad \Gamma \vdash N_t : B_t \qquad \Gamma \vdash N_f : B_f}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_t\ \texttt{else}\ N_f : B_t \sqcap B_f}$$

$$\textsc{GPair}\ \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2} \qquad \textsc{GPmPair}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : \pi_1(A), y : \pi_2(A) \vdash N : B}{\Gamma \vdash \texttt{let}\ (x, y)\ =\ M; N : B}$$

$$\textsc{GFun}\ \frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \textsc{GApp}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B \qquad \mathrm{dom}(A) \sim B}{\Gamma \vdash M\ N : \mathrm{cod}(A)}$$

$$\textsc{GPack}\ \frac{\Gamma, X \cong A \vdash M : B}{\Gamma \vdash \texttt{pack}^\nu(X \cong A, M) : \exists^\nu X.B} \qquad \textsc{GUnpack}\ \frac{\Gamma \vdash M : A \qquad \Gamma, X, x : \mathrm{un}\exists^\nu{}_X(A) \vdash N : B \qquad \Gamma \vdash B}{\Gamma \vdash \texttt{unpack}\ (X, x) = M; N : B}$$

$$\textsc{GPolyFun}\ \frac{\Gamma, X \vdash M : A}{\Gamma \vdash \Lambda^\nu X.M : \forall^\nu X.A} \qquad \textsc{GPolyApp}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B \qquad \Gamma, X \cong B, x : \mathrm{un}\forall^\nu{}_X(A) \vdash N : B_o \qquad \Gamma \vdash B_o}{\Gamma \vdash \texttt{let}\ x = M\{X \cong B\}; N : B_o}$$

$$? \sim A \qquad A \sim ? \qquad \texttt{Bool} \sim \texttt{Bool} \qquad X \sim X$$

$$\frac{A_i \sim B_i \qquad A_o \sim B_o}{A_i \to A_o \sim B_i \to B_o} \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \times A_2 \sim B_1 \times B_2} \qquad \frac{A \sim B}{\exists^\nu X.A \sim \exists^\nu X.B}$$

$$\frac{A \sim B}{\forall^\nu X.A \sim \forall^\nu X.B}$$

$$\begin{aligned}
\mathrm{dom}(A \to B) &= A & & \\
\mathrm{dom}(?) &= ? & A \sqcap ? &= A \\
\mathrm{cod}(A \to B) &= B & ? \sqcap B &= B \\
\mathrm{cod}(?) &= ? & X \sqcap X &= X \\
\pi_i(A_1 \times A_2) &= A_i & \texttt{Bool} \sqcap \texttt{Bool} &= \texttt{Bool} \\
\pi_i(?) &= ? & (A_1 \times A_2) \sqcap (B_1 \times B_2) &= (A_1 \sqcap B_1) \times (A_2 \sqcap B_2) \\
\mathrm{un}\exists^\nu{}_Y(\exists^\nu X.A) &= A[Y/X] & (A_i \to A_o) \sqcap (B_i \to B_o) &= (A_i \sqcap B_i) \to (A_o \sqcap B_o) \\
\mathrm{un}\exists^\nu{}_Y(?) &= ? & (\exists^\nu X.A) \sqcap (\exists^\nu X.B) &= \exists^\nu X.(A \sqcap B) \\
\mathrm{un}\forall^\nu{}_Y(\forall^\nu X.A) &= A[Y/X] & (\forall^\nu X.A) \sqcap (\forall^\nu X.B) &= \forall^\nu X.(A \sqcap B) \\
\mathrm{un}\forall^\nu{}_Y(?) &= ? & &
\end{aligned}$$

Figure 10.3: PolyG$^\nu$ Type System

$$
\begin{array}{llll}
\text{tags} & T & ::= & X \mid \texttt{Bool} \mid \times \mid \to \mid \exists^\nu \mid \forall^\nu \\
\text{terms} & M & ::= & x \mid \texttt{seal}_X M \mid \texttt{unseal}_X M \mid \texttt{is}(T)\texttt{?}\ M \mid \texttt{true} \mid \texttt{false} \\
& & \mid & \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \mid (M, M) \mid \texttt{let}\ (x, x) = M; M \\
& & \mid & M\ M \mid \lambda x : A.M \mid \texttt{pack}^\nu(\text{new} X, M) \mid \texttt{unpack}\ (X, x) = M; N \\
& & \mid & \Lambda^\nu X.M \mid \texttt{let}\ x = M\,\{\text{new} X\}; N \mid \texttt{let}\ x = M; M \\
\text{environment} & \Gamma & ::= & \cdot \mid \Gamma, x \mid \Gamma, \text{unknown}\ X \mid \Gamma, \text{known}\ X
\end{array}
$$

Figure 10.4: Dynamically Typed PolyG$^\nu$

to System F there is a side condition $\Gamma \vdash B$ that ensures the type of the continuation does not leak the abstract type variable X. Finally, we have universals. The introduction form (GPolyFun) is as a typical System F Λ. The polymorphic function elimination form GPolyApp is dual to the pack form: the scrutinee M is instantiated with a freshly created type $X \cong B$, and the result of the application is bound to a variable x. Then the newly created known type variable is bound in the continuation N in addition to the value x. Similar to the unpack form, ensuring that the type of the continuation does not leak the known type variable X.

10.3 A DYNAMICALLY TYPED VARIANT OF POLYG$^\nu$

Next, we present a syntax for the dynamically typed fragment of PolyG$^\nu$, which will demonstrate what dynamically typed features, the existential and universal types of PolyG$^\nu$correspond to. The existential types give a kind of generative module system, while the universal types give an exotic dual. Dynamically typed PolyG$^\nu$ is unusual for a dynamically typed language in that it includes second class "tag variables" X like in PolyG$^\nu$. Like how PolyG$^\nu$ has abstract type variables X and known type variables $X \cong A$, Dynamic PolyG$^\nu$ has *unkown* type variables unknown X and *known* type variables known X. When we elaborate to PolyG$^\nu$, these known type variables will be interpreted as $X \cong ?$. We can think of the difference between known and unknown type variables as a difference in capabilities: unknown type variables unknown X only allow for checking if a value has the tag X, and known type variables known X additionally allow for sealing and unsealing values at the type X.

We present the abstract syntax of Dynamically typed PolyG$^\nu$ in Figure 10.4 and the scope-checking rules in Figure 10.5. Most of the rules are typical for a dynamically typed λ-calculus, so we will only describe the non-standard ones. First, the tag checking rule (DCheckTag), is indexed by a *tag* T rather than a ground type G since the language doesn't have types per se. These tags, defined in Figure 10.4 include

all the basic forms of the language (booleans, products, etc) in addition to the fresh tags X, whether they are known or unknown. Next, values can be "sealed" and "unsealed" with a *known* tag variable known $X \in \Gamma$ (DSeal, DUnseal). Further down, we see where known and unknown type variables are actually produced. First, the pack rule (DPack) introduces a new known tag variable in the body of the "module". When unpacking a module (DUnpack), the tag name X is scoped as *unknown*, so the continuation cannot create directly create or use values with tag X. Next, the Church-style polymorphic functions (DPolyFun) abstract over an unknown type variable, whereas the elimination form (DPolyApp) creates a new known type variable that can be used by the continuation of the application.

These features enable data abstraction with no need for explicit typing. For instance, our flipper module from earlier would be written:

$$\texttt{pack}^{\nu}(\text{new}X, (\texttt{seal}_X\texttt{true}, ((\lambda x.\texttt{seal}_X(\texttt{NOT}(\texttt{unseal}_X x))), (\lambda x.\texttt{unseal}_X x))))$$

where $\texttt{NOT} = \lambda x.\texttt{if}\ x\ \texttt{then false else true}$.

This corresponds in PolyG$^{\nu}$ to using a pack instantiating X with ?, i.e.,

$$\texttt{pack}^{\nu}(X \cong ?, (\texttt{seal}_X\texttt{true}, ((\lambda x : X.\texttt{seal}_X(\texttt{NOT}(\texttt{unseal}_X x))), (\lambda x : X.\texttt{unseal}_X x))))$$

where $\texttt{NOT} = \lambda x\texttt{Bool.if}\ x\ \texttt{then false else true}$. and the free theorems are equally valid with that instantiation.

While the known and unknown type variables might seem an odd feature for a dynamically typed language, they should just be thought of as capabilities to perform certain operations. An unknown type variable X provides access to $\texttt{seal}_X$ and $\texttt{unseal}_X$, while a known type variable provides additional access to the tag checking function $\texttt{is}(X)?$. Since these are just functions, it is easy to then use ordinary functional programming to abstract over and pass these capabilities around as first class values in the dynamically typed language.

To give a semantics to the dynamically typed PolyG$^{\nu}$, we simply elaborate to PolyG$^{\nu}$ in Figure 10.6. As mentioned above, unknown type variables correspond to abstract type variables and known type variables X correspond to known type variables isomorphic to the dynamic type ($X \cong ?$). The remaining rules are straightforward, with all existential modules and polymorphic function instantiations using ?.

$$\textsc{DVar}\ \frac{x \in \Gamma}{\Gamma \vdash x} \qquad \textsc{DLet}\ \frac{\Gamma \vdash M \qquad \Gamma, x \vdash N}{\Gamma \vdash \texttt{let}\ x = M; N} \qquad \textsc{DCheckTag}\ \frac{\Gamma \vdash M \qquad \Gamma \vdash G}{\Gamma \vdash \texttt{is}(G)\texttt{?}\ M}$$

$$\textsc{DSeal}\ \frac{\Gamma \vdash M \qquad \text{known } X \in \Gamma}{\Gamma \vdash \texttt{seal}_X M} \qquad \textsc{DUnseal}\ \frac{\Gamma \vdash M \qquad \text{known } X \in \Gamma}{\Gamma \vdash \texttt{unseal}_X M}$$

$$\textsc{DTrue}\ \Gamma \vdash \texttt{true} \qquad \textsc{DFalse}\ \Gamma \vdash \texttt{false} \qquad \textsc{DIf}\ \frac{\Gamma \vdash M \qquad \Gamma \vdash N_t \qquad \Gamma \vdash N_f}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_t\ \texttt{else}\ N_f}$$

$$\textsc{DPair}\ \frac{\Gamma \vdash M_1 \qquad \Gamma \vdash M_2}{\Gamma \vdash (M_1, M_2)} \qquad \textsc{DPmPair}\ \frac{\Gamma \vdash M \qquad \Gamma, x, y \vdash N}{\Gamma \vdash \texttt{let}\ (x, y)\ =\ M; N}$$

$$\textsc{DFun}\ \frac{\Gamma, x \vdash M}{\Gamma \vdash \lambda x.M} \qquad \textsc{DApp}\ \frac{\Gamma \vdash M \qquad \Gamma \vdash N}{\Gamma \vdash M\ N}$$

$$\textsc{DPack}\ \frac{\Gamma, \text{known } X \vdash M : B}{\Gamma \vdash \texttt{pack}^{\nu}(\text{new} X, M)} \qquad \textsc{DUnpack}\ \frac{\Gamma \vdash M \qquad \Gamma, \text{unknown } X, x \vdash N}{\Gamma \vdash \texttt{unpack}\ (X, x) = M; N}$$

$$\textsc{DPolyFun}\ \frac{\Gamma, \text{unknown } X \vdash M}{\Gamma \vdash \Lambda^{\nu} X.M} \qquad \textsc{DPolyApp}\ \frac{\Gamma \vdash M \qquad \Gamma, \text{known } X, x \vdash N}{\Gamma \vdash \texttt{let}\ x = M\,\{\text{new} X\}; N}$$

Figure 10.5: Dynamic PolyG$^{\nu}$ Scope Checking

10.4 PolyC$^{\nu}$: cast calculus

As is standard in gradual languages, rather than giving the surface language an operational semantics directly, we define a *cast calculus* that makes explicit the casts that perform the dynamic type checking in gradual programs. We present the cast calculus syntax in Figure 10.7. The cast calculus syntax is almost the same as the surface syntax, with a few runtime-specific forms added, and the type annotation form replaced by two cast forms. First, we add *runtime type tags* σ to the types. These will be generated fresh at runtime and substituted in for type variables. We use α to refer to either a runtime type tag σ or a type variable X. Ground types G are the same as before except that they also include runtime type tags. Next, the terms remove the type annotation form, and in their stead adds upcasts $\langle A^{\sqsubseteq} \rangle\!\uparrow M$ and downcasts $\langle A^{\sqsubseteq} \rangle\!\downarrow M$, which are here annotated by a type precision

$$
\begin{aligned}
\lceil \cdot \rceil &= \cdot \\
\lceil \Gamma, x \rceil &= \lceil \Gamma \rceil, x : ? \\
\lceil \Gamma, \text{known } X \rceil &= \lceil \Gamma \rceil, X \cong ? \\
\lceil \Gamma, \text{unknown } X \rceil &= \lceil \Gamma \rceil, X
\end{aligned}
$$

$$
\begin{aligned}
\lceil X \rceil &= X \\
\lceil \texttt{Bool} \rceil &= \texttt{Bool} \\
\lceil \times \rceil &= ? \times ? \\
\lceil \to \rceil &= ? \to ? \\
\lceil \exists^{\nu} \rceil &= \exists^{\nu} X.? \\
\lceil \forall^{\nu} \rceil &= \forall^{\nu} X.?
\end{aligned}
$$

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil \texttt{let } x = M; N \rceil &= \texttt{let } x = \lceil M \rceil; \lceil N \rceil \\
\lceil \texttt{seal}_X M \rceil &= (\texttt{seal}_X \lceil M \rceil) :: ? \\
\lceil \texttt{unseal}_X M \rceil &= (\texttt{unseal}_X \lceil M \rceil) :: ? \\
\lceil \texttt{is(}T\texttt{)? } M \rceil &= \texttt{is(}\lceil T \rceil\texttt{)? } \lceil M \rceil \\
\lceil \texttt{true} \rceil &= \texttt{true} :: ? \\
\lceil \texttt{false} \rceil &= \texttt{false} :: ? \\
\lceil \texttt{if } M \texttt{ then } N_t \texttt{ else } N_f \rceil &= \texttt{if } \lceil M \rceil \texttt{ then } \lceil N_t \rceil \texttt{ else } \lceil N_f \rceil \\
\lceil (M_1, M_2) \rceil &= (\lceil M_1 \rceil, \lceil M_2 \rceil) :: ? \\
\lceil \texttt{let } (x, y) = M; N \rceil &= \texttt{let } (x, y) = \lceil M \rceil; \lceil N \rceil \\
\lceil \lambda x.M \rceil &= (\lambda x : ?.\lceil M \rceil) :: ? \\
\lceil M\ N \rceil &= \lceil M \rceil\ \lceil N \rceil \\
\lceil \texttt{pack}^{\nu}(\text{new} X, M) \rceil &= (\texttt{pack}^{\nu}(X \cong ?, \lceil M \rceil)) :: ? \\
\lceil \texttt{unpack } (X, x) = M; N \rceil &= \texttt{unpack } (X, x) = \lceil M \rceil; \lceil N \rceil \\
\lceil \Lambda^{\nu} X.M \rceil &= (\Lambda^{\nu} X.\lceil M \rceil) :: ? \\
\lceil \texttt{let } x = M\,\{\text{new} X\}; N \rceil &= \texttt{let } x = \lceil M \rceil \{X \cong ?\}; \lceil N \rceil
\end{aligned}
$$

Figure 10.6: Translation of Dynamic PolyG$^{\nu}$ into Gradual PolyG$^{\nu}$

$$
\begin{array}{rrcl}
\text{types} & A, B & + ::= & \sigma \\
\text{type names} & \alpha & ::= & \sigma \mid X \\
\text{ground types} & G & ::= & \alpha \mid \texttt{Bool} \mid ? \times ? \mid ? \to ? \mid \exists^{\nu} X.? \mid \forall^{\nu} X.? \\
\text{prec. derivations} & A^{\sqsubseteq}, B^{\sqsubseteq} & ::= & ? \mid \mathrm{tag}_G(A^{\sqsubseteq}) \mid \alpha \mid \texttt{Bool} \mid A^{\sqsubseteq} \times A^{\sqsubseteq} \\
 & & \mid & A^{\sqsubseteq} \to B^{\sqsubseteq} \mid \exists^{\nu} X.A^{\sqsubseteq} \mid \forall^{\nu} X.A^{\sqsubseteq} \\
\text{terms} & M, N & - ::= & (M :: A) \\
 & & + ::= & \mho \mid \langle A^{\sqsubseteq} \rangle\!\uparrow M \mid \langle A^{\sqsubseteq} \rangle\!\downarrow M \\
 & & \mid & \mid \texttt{seal}_{\sigma} M \mid \texttt{unseal}_{\sigma} M \mid \texttt{is}(\sigma)\texttt{?}\ M \\
 & & \mid & \texttt{pack}^{\nu}(X \cong A, \overline{[B^{\sqsubseteq} \updownarrow]}, M) \mid \Lambda^{\nu}\{X.(\overline{[B^{\sqsubseteq} \updownarrow]}, M)\} \\
\text{values} & V & ::= & \langle \mathrm{tag}_G(G) \rangle\!\uparrow V \mid \texttt{seal}_{\alpha} V \mid \texttt{true} \mid \texttt{false} \mid x \mid (V, V) \\
 & & \mid & \lambda(x : A).M \mid \langle A^{\sqsubseteq} \to B^{\sqsubseteq} \rangle\!\updownarrow V \\
 & & \mid & \Lambda^{\nu}\{X.(\overline{[B^{\sqsubseteq} \updownarrow]}, M)\} \mid \texttt{pack}^{\nu}(X \cong A, \overline{[B^{\sqsubseteq} \updownarrow]}, M) \\
\text{eval. contexts} & E & ::= & [] \mid (E, M) \mid (V, E) \mid E\ M \mid V\ E \\
 & & \mid & \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \texttt{let}\ (x, y) = E; M \\
 & & \mid & \langle A^{\sqsubseteq} \rangle\!\updownarrow E \mid \texttt{seal}_{\alpha} E \mid \texttt{unseal}_{\alpha} E \mid \texttt{is}(\alpha)\texttt{?}\ E \\
 & & \mid & \texttt{unpack}\ (X, x) = E; M \mid \texttt{let}\ x = E\{X \cong A\}; N
\end{array}
$$

Figure 10.7: PolyC$^{\nu}$ Syntax

derivation $A^{\sqsubseteq}$, which will be introduced shortly. Next, we also add seal, unseal and tag checking with runtime type tags. Finally, we introduce proxy forms for existential packages and polymorphic functions that collect a sequence of casts that will be run when the elimination form is used. We use the up-down arrow $\updownarrow$ is used to stand in for either an upcast arrow $\uparrow$ or downcast arrow $\downarrow$. So for example, the pack proxy form $\texttt{pack}^{\nu}(X \cong A, \overline{[B^{\sqsubseteq} \updownarrow]}, M)$ is essentially equivalent to the application of a sequence of existential type casts $\exists X.B^{\sqsubseteq}$ applied to a pack $\texttt{pack}^{\nu}(X \cong A, M)$. Similarly the polymorphic function proxy $\Lambda^{\nu}\{X.(\overline{[B^{\sqsubseteq} \updownarrow]}, M)\}$ is essentially equivalent to a sequence of casts applied to a polymorphic function $\Lambda^{\nu} X.M$. Next we have value forms, which include dynamically typed values, tagged with a ground type G, values sealed with a type name $\texttt{seal}_{\alpha} V$, booleans, variables, pairs, functions, proxied functions, proxied polymorphic functions, and proxied existential packages. Evaluation contexts are mostly standard for call-by-value, with the addition of casts, seals/unseals and runtime tag checks.

10.4.1 *PolyC$^{\nu}$ Type Precision*

Based on our analysis in Chapter 3, we add two cast forms: an upcast $\langle A^{\sqsubseteq} \rangle\!\uparrow M$ and a downcast $\langle A^{\sqsubseteq} \rangle\!\downarrow M$, whereas most prior work includes a single cast form $\langle A \Leftarrow B \rangle$. The $A^{\sqsubseteq}$ used in the upcast and downcast forms here is a proof that $A_l \sqsubseteq A_r$ for some types A_l, A_r, i.e., that

$$\frac{\Gamma \vdash A^{\sqsubseteq} : A \sqsubseteq G}{\Gamma \vdash \mathrm{tag}_G(A^{\sqsubseteq}) : A \sqsubseteq ?} \qquad \Gamma \vdash ? : ? \sqsubseteq ? \qquad \Gamma \vdash \mathtt{Bool} : \mathtt{Bool} \sqsubseteq \mathtt{Bool}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X : X \sqsubseteq X} \qquad \frac{\Gamma \vdash A_1^{\sqsubseteq} : A_{l1} \sqsubseteq A_{r1} \qquad \Gamma \vdash A_2^{\sqsubseteq} : A_{l2} \sqsubseteq A_{r2}}{\Gamma \vdash A_1^{\sqsubseteq} \times A_2^{\sqsubseteq} : A_{l1} \times A_{l2} \sqsubseteq A_{r1} \times A_{r2}}$$

$$\frac{\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r \qquad \Gamma \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma \vdash A^{\sqsubseteq} \to B^{\sqsubseteq} : A_l \to B_l \sqsubseteq A_r \to B_r}$$

$$\frac{\Gamma, X \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \exists^{\nu} X.A^{\sqsubseteq} : \exists^{\nu} X.A_l \sqsubseteq \exists^{\nu} X.A_r} \qquad \frac{\Gamma, X \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \forall^{\nu} X.A^{\sqsubseteq} : \forall^{\nu} X.A_l \sqsubseteq \forall^{\nu} X.A_r}$$

Figure 10.8: PolyC$^{\nu}$ Type Precision

A_l is a more precise (less dynamic) type than A_r. This type precision definition is key to formalizing the graduality property, but previous work has shown that it is useful for formalizing the semantics of casts as well. We emphasize the structure of these proofs because the central semantic constructions of this work: the operational semantics of casts, the translation of casts into functions and finally our graduality logical relation are all naturally defined by recursion on these derivations.

We present the definition of type precision in Figure 10.8. The judgment $\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$ is read as "using the variables in Γ", $A^{\sqsubseteq}$ proves that A_l is more precise (less dynamic) than A_r. If you ignore the precision derivations, our definition of type precision is a simple extension of the usual notion: type variables are only related to the dynamic type and themselves, and similarly for $\forall$ and $\exists$. Since we have quantifiers and type variables, we include a context Γ of known and abstract type variables. Crucially, even under the assumption that $X \cong A$, X and A are *unrelated* precision-wise unless A is ?. As before, $X \in \Gamma$ ranges over both known and abstract type variables. It is easy to see that precision reflexive and transitive, and that ? is the greatest element. Finally, ? is the least precise type, meaning for any type A there is a derivation that $A \sqsubseteq ?$. The precision notation is a natural extension of the syntax of types: with base types ?, `Bool` serving as the proof of reflexivity at the type and constructors $\times, \to$, etc. serving as syntax for congruence proofs. It is important to note that while we give a syntax for derivations, there is at most *one* derivation $A^{\sqsubseteq}$ that proves any given $A_l \sqsubseteq A_r$.

10.4.2 *PolyC$^{\nu}$ Type System*

The static type system for the cast calculus is given in Figure 10.9. The cast calculus type system differs from the surface language in that all type checking is *strict* and precise. This manifests in two ways. First,

the dynamic type is not considered implicitly compatible with other types. Instead, in the translation from PolyG$^\nu$ to PolyC$^\nu$, we insert casts wherever consistency is used in the judgment. Second, in the if rule, the branches must have the *same* type, and an upcast is inserted in the translation to make the two align.

Some of the semantics in Figure 10.11 involve terms with σs in places we would expect Xs, in particular instantiations, seals, and unseals. We also have our aforementioned intermediate form for `pack` and Λ^ν casts. Figure 10.9 gives the static typing rules for runtime terms. Note that the typing of runtime terms depends on a given Σ. To reason about well-typed terms at runtime, we also thread a store through the rules.

10.4.3 *Elaboration from PolyG$^\nu$ to PolyC$^\nu$*

We define the elaboration of PolyG$^\nu$ into the cast calculus PolyC$^\nu$ in Figure 10.10. Following [56], an ascription is interpreted as a cast up to ? followed by a cast down to the ascribed type. Most of the elaboration is standard, with elimination forms being directly translated to the corresponding PolyC$^\nu$ form if the head connective is correct, and inserting a downcast if the elimination position has type ?. We formalize this using the metafunction $G \wr M$ defined towards the bottom of the figure. For the if case, in PolyC$^\nu$ the two branches of the if have to have the same output type and export the same names, so we downcast each branch.

10.4.4 *PolyC$^\nu$ Operational Semantics*

The operational semantics of PolyC$^\nu$, presented in Figure 10.11, extends traditional cast semantics with appropriate rules for our name-generating universals and existentials. The runtime state is a pair of a term M and a *case store* Σ. A case store Σ represents the set of cases allocated so far in the program. Formally, a store Σ is just a pair of a number $\Sigma.n$ and a function $\Sigma.f : [n] \to \mathrm{Ty}$ where Ty is the set of all types and $[n] = \{m \in \mathbb{N} \mid m < n\}$ is from some prefix of natural numbers to types. All rules take configurations $\Sigma \triangleright M$ to configurations $\Sigma' \triangleright M'$. When the step does not change the store, we write $M \mapsto M'$ for brevity.

The first rule states that all non-trivial evaluation contexts propagate errors. Next, unsealing a seal gets out the underlying value, and `is(`G`)?` V literally checks if the tag of V is G. The pack and Λ^ν forms step to intermediate states used for building up a stack of casts that will be used again in the elimination rule. The unpack rule generates a fresh seal for the $X \cong A$ and then applies all of the accumulated casts to the body of the pack. Here we use $\updownarrow$ to indicate one of $\uparrow$ and $\downarrow$. The $\forall^\nu$ instantiation, rule is similar, generating a seal, casting the

$$\textsc{CErr}\ \frac{}{\Sigma;\Gamma \vdash \mho : A} \qquad \textsc{CVar}\ \frac{x : A \in \Gamma}{\Sigma;\Gamma \vdash x : A} \qquad \textsc{CLet}\ \frac{\Sigma;\Gamma \vdash M : A \qquad \Sigma;\Gamma, x : A \vdash N : B}{\Sigma;\Gamma \vdash \mathtt{let}\ x = M; N : B}$$

$$\textsc{CUpcast}\ \frac{\Sigma;\Gamma \vdash M : A_l \qquad \Sigma;\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Sigma;\Gamma \vdash \langle A^{\sqsubseteq}\rangle\!\uparrow M : A_r} \qquad \textsc{CDowncast}\ \frac{\Sigma;\Gamma \vdash M : A_r \qquad \Sigma;\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Sigma;\Gamma \vdash \langle A^{\sqsubseteq}\rangle\!\downarrow M : A_l}$$

$$\textsc{CCheckTy}\ \frac{\Sigma;\Gamma \vdash M : ? \qquad \Sigma;\Gamma \vdash G}{\Sigma;\Gamma \vdash \mathtt{is(}G\mathtt{)?}\ M : \mathtt{Bool}}$$

$$\textsc{CSeal}\ \frac{\Sigma;\Gamma \vdash M : A \qquad X \cong A \in \Gamma}{\Sigma;\Gamma \vdash \mathtt{seal}_X M : X} \qquad \textsc{CUnseal}\ \frac{\Sigma;\Gamma \vdash M : X \qquad X \cong A \in \Gamma}{\Sigma;\Gamma \vdash \mathtt{unseal}_X M : A}$$

$$\textsc{CSealRT}\ \frac{\Sigma;\Gamma \vdash M : A \qquad \sigma : A \in \Sigma}{\Sigma;\Gamma \vdash \mathtt{seal}_\sigma M : \sigma} \qquad \textsc{CUnsealRT}\ \frac{\Sigma;\Gamma \vdash M : \sigma \qquad \sigma : A \in \Sigma}{\Sigma;\Gamma \vdash \mathtt{unseal}_\sigma M : A}$$

$$\textsc{CTrue}\ \Sigma;\Gamma \vdash \mathtt{true} : \mathtt{Bool} \qquad \textsc{CFalse}\ \Sigma;\Gamma \vdash \mathtt{false} : \mathtt{Bool}$$

$$\textsc{CIf}\ \frac{\Sigma;\Gamma \vdash M : \mathtt{Bool} \qquad \Sigma;\Gamma \vdash N_t : B \qquad \Sigma;\Gamma \vdash N_f : B}{\Sigma;\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_t\ \mathtt{else}\ N_f : B}$$

$$\textsc{CPair}\ \frac{\Sigma;\Gamma \vdash M_1 : A_1 \qquad \Sigma;\Gamma \vdash M_2 : A_2}{\Sigma;\Gamma \vdash (M_1, M_2) : A_1 \times A_2} \qquad \textsc{CPmPair}\ \frac{\Sigma;\Gamma \vdash M : A_1 \times A_2 \qquad \Sigma;\Gamma, x : A_1, y : A_2 \vdash N : B}{\Sigma;\Gamma \vdash \mathtt{let}\ (x, y)\ =\ M; N : B}$$

$$\textsc{CFun}\ \frac{\Sigma;\Gamma \vdash A \qquad \Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda x : A.M : A \to B} \qquad \textsc{CApp}\ \frac{\Sigma;\Gamma \vdash M : A \to B \qquad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M\ N : B}$$

$$\textsc{CPack}\ \frac{\Sigma;\Gamma, X \cong A \vdash M : B}{\Sigma;\Gamma \vdash \mathtt{pack}^\nu(X \cong A, M) : \exists^\nu X.B} \qquad \textsc{CPackProxy}\ \frac{\Sigma;\Gamma, X \cong A' \vdash \overline{\langle A^{\sqsubseteq}\rangle\!\updownarrow} M : B}{\Sigma;\Gamma \vdash \mathtt{pack}^\nu(X \cong A', [\overline{A^{\sqsubseteq}\ \updownarrow}], M) : \exists^\nu X.B}$$

$$\textsc{CUnpack}\ \frac{\Sigma;\Gamma \vdash M : \exists^\nu X.A \qquad \Sigma;\Gamma, X, x : A \vdash N : B \qquad \Sigma;\Gamma \vdash B}{\Sigma;\Gamma \vdash \mathtt{unpack}\ (X, x) = M; N : B}$$

$$\textsc{CPolyFun}\ \frac{\Sigma;\Gamma, X \vdash M : A}{\Sigma;\Gamma \vdash \Lambda^\nu X.M : \forall^\nu X.A} \qquad \textsc{CPolyFunProxy}\ \frac{\Sigma;\Gamma, X \vdash \overline{\langle B^{\sqsubseteq}\rangle\!\updownarrow} M : A}{\Sigma;\Gamma \vdash \Lambda^\nu\{X.([\overline{B^{\sqsubseteq}\ \updownarrow}], M)\} : \forall^\nu X.A}$$

$$\textsc{CPolyApp}\ \frac{\Sigma;\Gamma \vdash M : \forall^\nu X.A \qquad \Sigma;\Gamma \vdash B \qquad \Sigma;\Gamma, X, x : A \vdash N : B_o \qquad \Sigma;\Gamma \vdash B_o}{\Sigma;\Gamma \vdash \mathtt{let}\ x = M\{X \cong B\}; N : B_o}$$

Figure 10.9: PolyC$^\nu$ Typing

$$
\begin{aligned}
(M :: B)^+ &= \langle B?^{\sqsubseteq}\rangle{\downarrow}\ \langle A?^{\sqsubseteq}\rangle{\uparrow}\ M^+ \\
&\qquad\qquad \text{(where } M : A,\ A?^{\sqsubseteq} : A \sqsubseteq ?,\ B?^{\sqsubseteq} : B \sqsubseteq ?\text{)} \\
x^+ &= x \\
(\texttt{let}\ x = M; N)^+ &= \texttt{let}\ x = M^+; N^+ \\
(\texttt{seal}_X M)^+ &= \texttt{seal}_X (M :: A)^+ \qquad \text{(where } X \cong A\text{)} \\
(\texttt{unseal}_X M)^+ &= \texttt{unseal}_X (X \wr M) \\
(\texttt{is(}G\texttt{)?}\ M)^+ &= \texttt{is(}G\texttt{)?}\ (\langle A?^{\sqsubseteq}\rangle{\uparrow}\ M) \\
&\qquad\qquad \text{(where } M : A,\ A?^{\sqsubseteq} : A \sqsubseteq ?\text{)} \\
b^+ &= b \qquad\qquad (b \in \{\texttt{true}, \texttt{false}\}) \\
(\texttt{if}\ M\ \texttt{then}\ N_t\ \texttt{else}\ N_f)^+ &= \texttt{if Bool} \wr M\ \texttt{then}\ (\langle B_t^{\sqsubseteq}\rangle{\downarrow}\ N_t^+) \\
&\qquad\qquad \texttt{else}\ (\langle B_f^{\sqsubseteq}\rangle{\downarrow}\ N_f^+) \\
&\qquad\qquad \text{(where } \texttt{if}\ M\ \texttt{then}\ N_t\ \texttt{else}\ N_f : B_t \sqcap B_f\text{)} \\
&\qquad\qquad \text{(and } B_t^{\sqsubseteq} : B_t \sqcap B_f \sqsubseteq B_t,\ B_f^{\sqsubseteq} : B_t \sqcap B_f \sqsubseteq B_f\text{)} \\
(M_1, M_2)^+ &= (M_1^+, M_2^+) \\
(\texttt{let}\ (x, y) = M; N)^+ &= \texttt{let}\ (x, y) = ? \times ? \wr M; N^+ \\
(\lambda x : A.M)^+ &= \lambda x : A.M^+ \\
(M\ N)^+ &= (? \to ? \wr M)\ (N :: \mathrm{dom}(A))^+ \\
&\qquad\qquad \text{(where } M : A\text{)} \\
(\texttt{pack}^{\nu}(X \cong A, M))^+ &= \texttt{pack}^{\nu}(X \cong A, M^+) \\
(\texttt{unpack}\ (X, x) = M; N)^+ &= \texttt{unpack}\ (X, x) = \exists^{\nu} X.? \wr M; N^+ \\
\Lambda^{\nu} X.M^+ &= \Lambda^{\nu} X.M^+ \qquad \text{(where } M : A\text{)} \\
\texttt{let}\ x = M\{X \cong A\}; N^+ &= \texttt{let}\ x = \forall^{\nu} Y.? \wr M\{X \cong A\}; N^+ \\
G \wr M &= \langle \mathrm{tag}_G(G)\rangle{\downarrow}\ M^+ \qquad \text{(when } M : ?\text{)} \\
G \wr M &= M^+ \qquad \text{(otherwise)}
\end{aligned}
$$

Figure 10.10: Elaborating PolyG$^{\nu}$ to PolyC$^{\nu}$

$$
\begin{array}{lcl}
E[\mho] & \mapsto & \mho \text{ where } E \neq [] \\
E[\texttt{let } x = V; M] & \mapsto & E[M[V/x]] \\
E[\texttt{unseal}_\sigma(\texttt{seal}_\sigma V)] & \mapsto & E[V] \\
E[\texttt{is(}G\texttt{)?}\ (\langle \text{tag}_G(G)\rangle\!\uparrow\ V)] & \mapsto & E[\texttt{true}] \\
E[\texttt{is(}G\texttt{)?}\ (\langle \text{tag}_H(H)\rangle\!\uparrow\ V)] & \mapsto & E[\texttt{false}] \text{ where } G \neq H \\
\Sigma \rhd E\left[\begin{array}{l}\texttt{let } x = (\Lambda^\nu\{X.([\overline{B^{\sqsubseteq}}\updownarrow], M)\})\{Y \cong A\}; \\ N\end{array}\right] & \mapsto & \Sigma, \sigma : A \rhd E\left[\begin{array}{l}\texttt{let } y = \overline{\langle B^{\sqsubseteq}[\sigma/X]\rangle\!\updownarrow} M[\sigma/X]; \\ N[\sigma/Y]\end{array}\right] \\
\Sigma \rhd E\left[\begin{array}{l}\texttt{unpack } (Y, x) = \texttt{pack}^\nu(X \cong A, [\overline{B^{\sqsubseteq}}\updownarrow], M); \\ N\end{array}\right] & \mapsto & \Sigma, \sigma : A \rhd E\left[\begin{array}{l}\texttt{let } y = \overline{\langle B^{\sqsubseteq}[\sigma/X]\rangle\!\updownarrow} M[\sigma/X]; \\ N[\sigma/Y]\end{array}\right] \\
E\left[\texttt{pack}^\nu(X \cong A, M)\right] & \mapsto & E[\texttt{pack}^\nu(X \cong A', [], M)] \\
E[(\lambda(x : A).M)\ V_a] & \mapsto & E[M[V_a/x]] \\
E[(\langle A^{\sqsubseteq} \to B^{\sqsubseteq}\rangle\!\updownarrow\ V_f)\ V_a] & \mapsto & E[\langle B^{\sqsubseteq}\rangle\!\updownarrow\ (V_f(\langle A^{\sqsubseteq}\rangle\!\updownarrow^-\ V_a))]E[M[V/x]] \\
E[\texttt{if true then } M_1 \texttt{ else } M_2] & \mapsto & E[M_1] \\
E[\texttt{if false then } M_1 \texttt{ else } M_2] & \mapsto & E[M_2] \\
E[\texttt{let } (x, y) = (V_1, V_2); M] & \mapsto & E[M[V_1/x][V_2/y]] \\
E[\langle A^{\sqsubseteq}\rangle\!\updownarrow\ V] & \mapsto & E[V] \text{ where } A^{\sqsubseteq} \in \{\texttt{Bool}, \sigma, ?\} \\
E[\langle A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}\rangle\!\updownarrow\ (V_1, V_2)] & \mapsto & E[(\langle A_1^{\sqsubseteq}\rangle\!\updownarrow\ V_1, \langle A_2^{\sqsubseteq}\rangle\!\updownarrow\ V_2)] \\
E[\langle \text{tag}_G(A^{\sqsubseteq})\rangle\!\uparrow\ V] & \mapsto & E[\langle \text{tag}_G(G)\rangle\!\uparrow\ \langle A^{\sqsubseteq}\rangle\!\uparrow\ V] \text{ when } A^{\sqsubseteq} \neq G \\
E[\langle \text{tag}_G(A^{\sqsubseteq})\rangle\!\downarrow\ \langle \text{tag}_G(G)\rangle\!\uparrow\ V] & \mapsto & E[\langle A^{\sqsubseteq}\rangle\!\downarrow\ V] \\
E[\langle \text{tag}_G(A^{\sqsubseteq})\rangle\!\downarrow\ \langle \text{tag}_H(H)\rangle\!\uparrow\ V] & \mapsto & \mho \text{ where } H \neq G \\
E[\langle \exists^\nu X.A^{\sqsubseteq}\rangle\!\updownarrow\ \texttt{pack}^\nu(Y \cong A', [\overline{B^{\sqsubseteq}}\updownarrow], M)] & \mapsto & E[\texttt{pack}^\nu(Y \cong A', [A^{\sqsubseteq}[Y/X]\updownarrow, \overline{B^{\sqsubseteq}}\updownarrow], M)] \\
E[\langle \forall^\nu X.A^{\sqsubseteq}\rangle\!\updownarrow\ \Lambda^\nu\{Y.([\overline{B^{\sqsubseteq}}\updownarrow], M)\}] & \mapsto & E[\Lambda^\nu\{Y.([A^{\sqsubseteq}[Y/X]\updownarrow, \overline{B^{\sqsubseteq}}\updownarrow], M)\}]
\end{array}
$$

Figure 10.11: PolyC$^\nu$ Operational Semantics

function and substituting the seal in. As is typical for a cast calculus, the remaining types have ordinary call-by-value β reductions.

The remaining rules give the behavior of casts. Other than the use of type precision derivations, the behavior of our casts is mostly standard: identity casts for `Bool`, σ and ? are just the identity, and the product cast proceeds structurally. Function casts are values, and when applied to a value, the cast is performed on the output and the oppositely oriented case on the input. We use $\updownarrow^-$ to indicate the opposite arrow, so $\uparrow^- = \downarrow$ and $\downarrow^- = \uparrow$ to cut down the number of rules. Next, the $\forall^\nu$ casts are also values that reduce when the instantiating type is supplied. As with existentials, the freshly generated type σ is substituted for X in the precision derivation guiding the cast. Finally, the upcast case for $\text{tag}_G(A^{\sqsubseteq})$ simply injects the result of upcasting with $A^{\sqsubseteq}$ into the dynamic type using the tag G. For the downcast case, the opposite is done if the input has the right tag, and otherwise a dynamic type error is raised.

We mention a few standard operational lemmas that are easily verified.

Lemma 188 (Unique Decomposition). *If* $\Sigma_1; \cdot \vdash M_1 : A$*, then there exist unique* E, M_2 *such that* $M_1 = E[M_2]$.

Lemma 189 (Cast calculus dynamic semantics are deterministic). *If* $\Sigma \triangleright M \mapsto \Sigma_1 \triangleright M_1$ *and* $\Sigma \triangleright M \mapsto \Sigma_2 \triangleright M_2$ *then* $\Sigma_1 = \Sigma_2$ *and* $M_1 = M_2$.

Lemma 190 (Progress). *If* $\Sigma_1; \cdot \vdash M_1 : A$ *then either* $\Sigma_1 \triangleright M_1 \mapsto \Sigma_2 \triangleright M_2$, $M_1 = \mathcal{O}$, *or* $M_1 = V$ *for some* V.

10.5 TYPED INTERPRETATION OF THE CAST CALCULUS

In the previous section we developed a cast calculus with an operational semantics defining the behavior of the name generation and gradual type casts. However, this ad hoc design addition of new type connectives and complex cast forms make the cast calculus less than ideal for proving meta-theoretic properties of the system.

Instead of directly proving metatheoretic properties of the cast calculus, we give a *contract translation* of the cast calculus into a statically typed core language, translating the gradual type casts to ordinary terms in the typed language that raise errors. The key benefit of the typed language is that it does not have built-in notions of fresh existential and universal quantification. Instead, the type translation decomposes these features into the combination of *ordinary* existential and universal quantification combined with a somewhat well-studied programming feature: a dynamically extensible "open" sum type we call OSum. Finally, it gives a static type interpretation of the dynamic type: rather than being a finitary sum of a few statically fixed cases, the dynamic type is implemented as the open sum type which includes those types allocated at runtime.

10.5.1 *Typed Metalanguage*

We present the syntax of our typed language $\text{CBPV}_{\text{OSum}}$ in Figure 10.12, an extension of Levy's Call-by-push-value calculus [45], which we use as a convenient metalanguage to extend with features of interest. We utilized Call-by-push-value (CBPV) extensively in Part II, but we review it quickly now to make this chapter more independently readable. CBPV is a typed calculus with highly explicit evaluation order, providing similar benefits to continuation-passing style and A-normal form [65]. The main distinguishing features of CBPV are that values V and effectful computations M are distinct syntactic categories, with distinct types: value types A and computation types B. The two "shift" types U and F mediate between the two worlds. A value of type UB is a first-class "thUnk" of a computation of type B that can be forced,

$$
\begin{array}{rrcl}
\text{value types} & A & ::= & X \mid \mathsf{Case}\ A \mid \mathsf{OSum} \mid A \times A \mid \mathtt{Bool} \mid \exists X.A \mid UB \\
\text{computation types} & B & ::= & A \to B \mid \forall X.B \mid FA \\
\text{values} & V & ::= & \sigma \mid \mathtt{inj}_V\ V \mid \mathtt{pack}(A, V)\ \mathtt{as}\ \exists X.A \mid x \mid (V, V) \\
& & \mid & \mathtt{true} \mid \mathtt{false} \mid \mathtt{thunk}\ M \\
\text{computations} & M & ::= & \mho \mid \mathtt{force}\ V \mid \mathtt{ret}\ V \mid x \leftarrow M; N \mid M\ V \mid \lambda x : A.M \\
& & \mid & \mathtt{newcase}_A\ x; M \mid \mathtt{match}\ V\ \mathtt{with}\ V\{\mathtt{inj}\ x.M \mid N\} \\
& & \mid & \mathtt{unpack}\ (X, x) = V; M \mid \mathtt{let}\ (x, y)\ =\ V; M \\
& & \mid & \Lambda X.M \mid M[A] \mid \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ M \\
\text{stacks} & S & ::= & \bullet \mid S\ V \mid S\ [A] \mid x \leftarrow S; M \\
\text{value typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{type variable context} & \Delta & ::= & \cdot \mid \Delta, X
\end{array}
$$

Figure 10.12: $\text{CBPV}_{\text{OSum}}$ Syntax

behaving as a B. A computation of type FA is a computation that can perform effects and return a value of type A, and whose elimination form is a monad-like *bind*. Notably while sums and (strict) tuples are value types, function types $A \to B$ are computations since a function interacts with its environment by receiving an argument. We include existentials as value type and universals as computation types, that in each case quantify over *value* types because we are using it as the target of a translation from a call-by-value language.

We furthermore extend CBPV with two new value types: OSum and Case A, which add an open sum type similar to the extensible exception types in ML, but with an expression-oriented interface more suitable to a core calculus. The open sum type OSum is initially empty, but can have new cases allocated at runtime. A value of Case A is a first class representative of a case of OSum. The introduction form $\mathtt{inj}_{V_c}\ V$ for OSum uses a case V_c : Case A to inject a value $V : A$ into OSum. The elimination form $\mathtt{match}\ V_o\ \mathtt{with}\ V_c\{\mathtt{inj}\ x.M \mid N\}$ for OSum is to use a V_c : Case A to do a pattern match on a value V_o : OSum. Since OSum is an *open* sum type, it is unknown what cases V_o might use, so the pattern-match has two branches: the one $\mathtt{inj}\ x.M$ binds the underlying value to $x : A$ and proceeds as M and the other is a catch-all case N in case V_o was not constructed using V_c. Finally, there is a form $\mathtt{newcase}_A\ x; M$ that allocates a fresh Case A, binds it to x and proceeds as M. In addition to the similarity to ML exception types, they are also similar to the dynamically typed sealing mechanism introduced in Sumii and Pierce [77].

10.5.2 *Static and Dynamic Semantics*

We show the typing rules for $\text{CBPV}_{\text{OSum}}$ in Figure 10.13. There are two judgments corresponding to the two syntactic categories of terms: $\Delta; \Gamma \vdash V : A$ for typing a value and $\Delta; \Gamma \vdash M : A$ for typing a

computation. Δ is the environment of type variables and Γ is the environment for term variables. Unlike in PolyG$^\nu$ and PolyC$^\nu$, these are completely standard, and there is no concept of a known type variable.

First, an error $\mho$ is a computation and can be given any type. Variables are standard and the OSum/Case forms are as described above. Existentials are a value form and are standard as in CBPV using ordinary substitution in the pack form. In all of the value type elimination rules, the discriminee is restricted to be a *value*. A computation $M : B$ can be thunked to form a value $\mathtt{thunk}\, M : UB$, which can be forced to run as a computation. Like the existentials, the universal quantification type is standard, using substitution in the elimination form. Finally, the introduction form for FA returns a value $V : A$, and the elimination form is a bind, similar to a monadic semantics of effects, except that the continuation can have any computation type B, rather than restricted to FA.

The operational semantics is given in Figure 10.14. S represents a *stack*, the CBPV analogue of an evaluation context, defined in Figure 10.12. Here Σ is like the Σ in PolyC$^\nu$, but maps to *value types*. The semantics is standard, other than the fact that we assign a count to each step of either 0 or 1. The only steps that count for 1 are those that introduce non-termination to the language, which is used later as a technical device in our logical relation in §10.6.

Note that this is a simple deterministic operational semantics.

Lemma 191 (target language operational semantics are deterministic)**.** *If* $\Sigma' \triangleright M' \mapsto \Sigma'_1 \triangleright M'_1$ *and* $\Sigma' \triangleright M' \mapsto \Sigma'_2 \triangleright M'_2$ *then* $\Sigma'_1 = \Sigma'_2$ *and* $M'_1 = M'_2$.

10.5.3 *Translation*

Next, we present the "contract translation" of PolyC$^\nu$ into CBPV$_{\text{OSum}}$. This translation can be thought of as an alternate semantics to the operational semantics for PolyC$^\nu$, but with a tight correspondence given in §10.5.4. Since CBPV$_{\text{OSum}}$ is a typed language that uses ordinary features like functions, quantification and an open sum type, this gives a simple explanation of the semantics of PolyC$^\nu$ in terms of fairly standard language features.

In the left side of Figure 10.15, we present the type translation from PolyC$^\nu$ to CBPV$_{\text{OSum}}$. Since PolyC$^\nu$ is a call-by-value language, types are translated to CBPV$_{\text{OSum}}$ *value* types. Booleans and pairs are translated directly, and the function type is given the standard CBPV translation for call-by-value functions, $U(\llbracket A \rrbracket \to F\llbracket B \rrbracket)$: a call-by-value function is a thunked computation of a function that takes an $\llbracket A \rrbracket$ as input and may return a $\llbracket B \rrbracket$ as output. The dynamic type ? is interpreted as the open sum type. The meaning of a type variable depends on the context: if it is an abstract type variable, it is translated

$$\frac{\Sigma;\Delta;\Gamma \vdash V_T : \text{Case } A \qquad \Sigma;\Delta;\Gamma \vdash V : \text{OSum} \qquad \Sigma;\Delta;\Gamma, x : A \mid \cdot \vdash M : B \qquad \Sigma;\Delta;\Gamma \mid \cdot \vdash N : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{match } V_T \texttt{ with } V\{\texttt{inj } x.M \mid N\} : B}$$

$$\frac{\sigma : A \in \Sigma \qquad \Delta \vdash A}{\Sigma;\Delta;\Gamma \vdash \sigma : \text{Case } A} \qquad \frac{\Delta \vdash A \qquad \Sigma;\Delta;\Gamma, x : \text{Case } A \mid \cdot \vdash M : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{newcase}_A\ x; M : B}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V_T : \text{Case } A \qquad \Sigma;\Delta;\Gamma \vdash V : A}{\Sigma;\Delta;\Gamma \vdash \texttt{inj}_{V_T}\ V : \text{OSum}}$$

$$\frac{\Sigma;\Delta, X;\Gamma \mid \cdot \vdash M : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \Lambda X.M : \forall X.B} \qquad \frac{\Sigma;\Delta;\Gamma \mid \Theta \vdash M : \forall X.B \qquad \Delta \vdash A}{\Sigma;\Delta;\Gamma \mid \Theta \vdash M[A] : B[A/X]}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V : A[A'/X]}{\Sigma;\Delta;\Gamma \vdash \texttt{pack}(A', V)\ \texttt{as}\ \exists X.A : \exists X.A}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V : \exists X.A \qquad \Delta \vdash B \qquad \Sigma;\Delta, X;\Gamma, x : A \mid \cdot \vdash M : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{unpack } (X, x) = V; M : B}$$

$$\frac{}{\Sigma;\Delta;\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Sigma;\Delta;\Gamma \mid \bullet : B \vdash \bullet : B}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V : \texttt{Bool} \qquad \Sigma;\Delta;\Gamma \mid \cdot \vdash M_1 : B \qquad \Sigma;\Delta;\Gamma \mid \cdot \vdash M_2 : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{if } V \texttt{ then } M_1 \texttt{ else } M_2 : B}$$

$$\frac{}{\Sigma;\Delta;\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Sigma;\Delta;\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V_1 : A_1 \qquad \Sigma;\Delta;\Gamma \vdash V_2 : A_2}{\Sigma;\Delta;\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V : A_1 \times A_2 \qquad \Sigma;\Delta;\Gamma, x : A_1, y : A_2 \mid \cdot \vdash M : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{let } (x, y) = V; M : B}$$

$$\frac{\Sigma;\Delta;\Gamma \mid \cdot \vdash M : B}{\Sigma;\Delta;\Gamma \vdash \texttt{thunk } M : UB} \qquad \frac{\Sigma;\Delta;\Gamma \vdash V : UB}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{force } V : B}$$

$$\frac{\Sigma;\Delta;\Gamma \vdash V : A}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \texttt{ret } V : FA} \qquad \frac{\Sigma;\Delta;\Gamma \mid \Theta \vdash M : FA \qquad \Sigma;\Delta;\Gamma \mid \cdot \vdash N : B}{\Sigma;\Delta;\Gamma \mid \Theta \vdash x \leftarrow M; N : B}$$

$$\frac{\Sigma;\Delta;\Gamma \mid \Theta \vdash M : A \to B \qquad \Sigma;\Delta;\Gamma \vdash V : A}{\Sigma;\Delta;\Gamma \mid \Theta \vdash M\ V : B}$$

$$\frac{\Delta \vdash A \qquad \Sigma;\Delta;\Gamma, x : A \mid \cdot \vdash V : B}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \lambda x : A.V : A \to B} \qquad \frac{}{\Sigma;\Delta;\Gamma \mid \cdot \vdash \mho : B}$$

Figure 10.13: CBPV Type System

$$
\begin{aligned}
S[\mathcal{U}] &\mapsto \mathcal{U} \\
\Sigma \triangleright S[\mathtt{newcase}_A\ x; M] &\mapsto \Sigma, \sigma : A \triangleright S[M[\sigma/x]] \\
S[\mathtt{match}\ \mathtt{inj}_\sigma\ V\ \mathtt{with}\ \sigma\{\mathtt{inj}\ x.M \mid N\}] &\mapsto S[M[V/x]] \\
S[\mathtt{match}\ \mathtt{inj}_{\sigma_1}\ V\ \mathtt{with}\ \sigma_2\{\mathtt{inj}\ x.M \mid N\}] &\mapsto S[N] \\
& \qquad (\text{where } \sigma_1 \neq \sigma_2) \\
S[\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ M\ \mathtt{else}\ N] &\mapsto S[M] \\
S[\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ M\ \mathtt{else}\ N] &\mapsto S[N] \\
S[\mathtt{let}\ (x,y) = (V_1, V_2); M] &\mapsto S[M[V_1/x, V_2/y]] \\
S[\mathtt{force}\ (\mathtt{thunk}\ M)] &\mapsto S[M] \\
S[\mathtt{unpack}\ (X,x) = \mathtt{pack}(A,V); M] &\mapsto S[M[A/X, V/x]] \\
S[(\Lambda X.M)[A]] &\mapsto S[M[A/X]] \\
S[(\lambda(x:A).M)\ V] &\mapsto S[M[V/x]] \\
S[x \leftarrow \mathtt{ret}\ V; N] &\mapsto S[N[V/x]]
\end{aligned}
$$

Figure 10.14: CBPV Operational Semantics

to a type variable, but if it is a known type variable $X \cong A$, it is translated to $[\![A]\!]$! That is, at runtime, values of a known type variable X are just values of the type isomorphic to X, and as we will see later, sealing and unsealing are no-ops. Similarly, a runtime type tag σ is translated to the type that the corresponding case maps to. These are inductively well-defined because Σ stays constant in the type translation and Γ only adds abstract type variables.

The final two cases are the most revealing. First the fresh universal quantifier, $\forall^\nu X.A$, translates to *not just* a thunk of a universally quantified computation, but also takes in a Case X as input. The body of a Λ will then use that Case X in order to interpret casts involving X. This is precisely why parametricity is more complex for our source language: if it were translated to just $U(\forall X.F[\![A]\!])$, then parametricity would follow directly by parametricity for $\text{CBPV}_{\text{OSum}}$, but the Case X represents additional information that the function is being passed that potentially provides information about the type X. It is only because code translated from PolyC^ν always generates a fresh case that this extra input is benign. The fresh existential $\exists^\nu X.A$ is translated to a real existential of a thunk that expects a Case X and returns a $[\![A]\!]$. Note that while the quantification is the dual of the $\forall^\nu$ case, both of them receive a Case X from the environment, which is freshly generated.

Next, while PolyG^ν and PolyC^ν have a single environment Γ that includes type variables and term variables, in $\text{CBPV}_{\text{OSum}}$, these are separated into a type variable environment Δ and a term variable

$$
\begin{aligned}
[\![\Sigma; \Gamma \vdash ?]\!] &= \mathrm{OSum} \\
[\![\Sigma; \Gamma \vdash X]\!] &= X \quad (\text{where } X \in \Gamma) \\
[\![\Sigma; \Gamma \vdash X]\!] &= [\![A]\!] \quad (\text{where } X \cong A \in \Gamma) \\
[\![\Sigma; \Gamma \vdash \sigma]\!] &= [\![A]\!] \quad (\text{where } \sigma : A \in \Sigma) \\
[\![\Sigma; \Gamma \vdash \mathtt{Bool}]\!] &= \mathtt{Bool} \\
[\![\Sigma; \Gamma \vdash A \to B]\!] &= U([\![\Sigma; \Gamma \vdash A]\!] \to F[\![\Sigma; \Gamma \vdash B]\!]) \\
[\![\Sigma; \Gamma \vdash A_1 \times A_2]\!] &= [\![\Sigma; \Gamma \vdash A_1]\!] \times [\![\Sigma; \Gamma \vdash A_2]\!] \\
[\![\Sigma; \Gamma \vdash \exists^{\nu} X.A]\!] &= \exists X.U(\mathrm{Case}\ X \to F[\![\Sigma; \Gamma, X \vdash A]\!]) \\
[\![\Sigma; \Gamma \vdash \forall^{\nu} X.A]\!] &= U(\forall X.\mathrm{Case}\ X \to F[\![\Sigma; \Gamma, X \vdash A]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![\Sigma \vdash \cdot]\!] &= \cdot; \cdot \\
[\![\Sigma \vdash \Gamma, x : A]\!] &= \Delta'; \Gamma', x : [\![\Sigma; \Gamma \vdash A]\!] \\
&\quad (\text{where } [\![\Sigma \vdash \Gamma]\!] = \Delta'; \Gamma') \\
[\![\Sigma \vdash \Gamma, X]\!] &= \Delta', X; \Gamma', c_X : \mathrm{Case}\ X \\
&\quad (\text{where } [\![\Sigma \vdash \Gamma]\!] = \Delta'; \Gamma') \\
[\![\Sigma \vdash \Gamma, X \cong A]\!] &= \Delta'; \Gamma', c_X : \mathrm{Case}\ [\![\Sigma; \Gamma \vdash A]\!] \\
&\quad (\text{where } [\![\Sigma \vdash \Gamma]\!] = \Delta'; \Gamma')
\end{aligned}
$$

Figure 10.15: PolyC$^{\nu}$ type and environment translation

environment Γ. For this reason in the right side of Figure 10.15 we define the translation of an environment Γ to be a pair of environments in CBPV$_{\mathrm{OSum}}$. The term variable $x : A$ is just translated to a variable $x : [\![A]\!]$, but the type variables are more interesting. An abstract type variable X is translated to a pair of a type variable X but also an associated term variable $c_X : \mathrm{Case}\ X$, which represents the case of the dynamic type that will be instantiated with a freshly generated case. On the other hand, since known type variables $X \cong A$ are translated to $[\![A]\!]$, we do not extend Δ with a new variable, but still produce a variable $c_X : [\![A]\!]$ as with an unknown type variable. Finally, the empty context $\cdot$ is translated to a pair of empty contexts.

To translate a whole program, written $[\![\Sigma; \cdot \vdash M]\!]_p$, we insert a preamble that generates the cases of the open sum type for each ground type. In Figure 10.16, we show our whole-program translation which inserts a preamble to generate a case of the OSum type for each ground type. This allows us to assume the existence of these cases in the rest of the translation. These can be conveniently modeled as a sequence of "global" definitions of some known type variables, which we write as Γ_p. We also define a function $\mathrm{case}(\cdot)$ from types to their corresponding case value, which is a case variable for all types except those generated at runtime σ.

For later convenience, we also define the store and substitution that are generated when this preamble runs.

$$
\begin{array}{rcl}
[\![M]\!]_p & = & \mathtt{newcase}_{[\![\mathtt{Bool}]\!]}\ c_{\text{Bool}}; \\
& & \mathtt{newcase}_{[\![?\to ?]\!]}\ c_{\text{Fun}}; \\
& & \mathtt{newcase}_{[\![?\times ?]\!]}\ c_{\text{Times}}; \\
& & \mathtt{newcase}_{[\![\exists^\nu X.?]\!]}\ c_{\text{Ex}}; \\
& & \mathtt{newcase}_{[\![\forall^\nu X.?]\!]}\ c_{\text{All}}; \\
& & [\![M]\!]
\end{array}
\qquad
\begin{array}{rcl}
\text{case}(\mathtt{Bool}) & = & c_{\text{Bool}} \\
\text{case}(A \to B) & = & c_{\text{Fun}} \\
\text{case}(A \times B) & = & c_{\text{Times}} \\
\text{case}(\exists^\nu X.A) & = & c_{\text{Ex}} \\
\text{case}(\forall^\nu X.A) & = & c_{\text{All}} \\
\text{case}(X) & = & c_X \\
\text{case}(\sigma) & = & \sigma
\end{array}
$$

$$\Gamma_p \quad = \quad \text{Bool} \cong \mathtt{Bool}, \text{Fun} \cong ? \to ?, \text{Times} \cong ? \times ?, \text{Ex} \cong \exists^\nu X.?, \text{All} \cong \forall^\nu X.?$$

Figure 10.16: Ground type tag management

Definition 192 (Preamble store, substitution)**.** We name the store generated by the preamble Σ_p, defined as

$$
\begin{array}{rcl}
\Sigma_p & = & (4, f) \\
f(0) & = & \mathtt{Bool} \\
f(1) & = & U(\text{OSum} \to F\text{OSum}) \\
f(2) & = & \text{OSum} \times \text{OSum} \\
f(3) & = & \exists X.U(\text{Case } X \to F\text{OSum}) \\
f(4) & = & U(\forall X.\text{Case } X \to F\text{OSum})
\end{array}
$$

We define γ_p to be a substitution that closes terms with respect to Γ_p using the store Σ_p:

$$
\begin{array}{rcl}
\gamma_p(c_{\text{Bool}}) & = & 0 \\
\gamma_p(c_{\text{Fun}}) & = & 1 \\
\gamma_p(c_{\text{Times}}) & = & 2 \\
\gamma_p(c_{\text{Ex}}) & = & 3 \\
\gamma_p(c_{\text{All}}) & = & 4
\end{array}
$$

Next, we consider the term translation, which is defined with the below type preservation Theorem 193 in mind. First, all PolyC$^\nu$ terms of type A are translated to CBPV$_{\text{OSum}}$ *computations*, with type $F[\![A]\!]$, which is standard for translating CBV to CBPV. Finally, we include the preamble context Γ_p to the front of the terms to account for the fact that all terms can use the cases generated in the preamble.

Theorem 193. *If* $\Gamma_1 \vdash M : A, \Gamma_2$ *then* $\Delta; \Gamma \vdash [\![M]\!] : F[\![\Sigma; \Gamma_1, \Gamma_2 \vdash A]\!]$ *where* $[\![\Gamma_p, \Gamma_1, \Gamma_2]\!] = \Delta; \Gamma$.

We show the term translation in Figures 10.17 and 10.18. To reduce the context clutter in the translations, we elide the contexts Σ, Γ in the definition of the semantics. While they are technically needed to translate type annotations, they do not affect the operational semantics and so can be safely ignored.

$$
\begin{array}{lcl}
[\![x]\!] & = & \mathtt{ret}\ x \\
[\![\mathtt{let}\ x = M; N]\!] & = & x \leftarrow [\![M]\!]; [\![N]\!] \\
[\![\mho_A]\!] & = & \mho \\
[\![\mathtt{seal}_\alpha M]\!] & = & [\![M]\!] \\
[\![\mathtt{unseal}_\alpha M]\!] & = & [\![M]\!] \\
[\![\mathtt{is}(G)?\ M]\!] & = & r \leftarrow [\![M]\!]; \\
 & & \mathtt{match}\ r\ \mathtt{with}\ \mathrm{case}(G)\{\mathtt{inj}\ y.\mathtt{ret}\ \mathtt{true} \mid \mathtt{ret}\ \mathtt{false}\} \\
[\![\langle A^{\sqsubseteq}\rangle\!\updownarrow\ M]\!] & = & [\![A^{\sqsubseteq}]\!]\!\updownarrow [\![M]\!] \\
[\![\mathtt{true}]\!] & = & \mathtt{ret}\ \mathtt{true} \\
[\![\mathtt{false}]\!] & = & \mathtt{ret}\ \mathtt{false} \\
[\![\mathtt{if}\ M_1\ \mathtt{then}\ M_2\ \mathtt{else}\ M_3]\!] & = & r \leftarrow [\![M_1]\!]; \mathtt{if}\ r\ \mathtt{then}\ [\![M_2]\!]\ \mathtt{else}\ [\![M_3]\!] \\
[\![(M_1, M_2)]\!] & = & x_1 \leftarrow [\![M_1]\!]; x_2 \leftarrow [\![M_2]\!]; \mathtt{ret}\ (x_1, x_2) \\
[\![\mathtt{let}\ (x,y)\ =\ M; N]\!] & = & r \leftarrow [\![M]\!]; \mathtt{let}\ (x,y)\ =\ r; [\![N]\!] \\
[\![\mathtt{pack}^\nu(X \cong A, M)]\!] & = & \mathtt{ret}\ \mathtt{pack}(A, \mathtt{thunk}\ (\lambda c_X : \mathrm{Case}\ A. [\![M]\!])) \\
[\![\mathtt{pack}^\nu(X \cong A', [A_n^{\sqsubseteq}\updownarrow_n \ldots A_1^{\sqsubseteq}\updownarrow_1], M)]\!] & = & \mathtt{ret}\ \mathtt{pack}(A, \mathtt{thunk}\ (\lambda c_X : \mathrm{Case}\ A. M'_n)) \\
 & & \text{where } M'_0 = [\![M]\!] \\
 & & \text{and } M'_{i+1} = \\
 & & [\![A_{i+1}^{\sqsubseteq}]\!]\!\updownarrow_{i+1} [\mathtt{force}\ (\mathtt{thunk}\ (\lambda c_X : \mathrm{Case}\ A'. [\![M_i]\!]))\ c_X] \\
[\![\mathtt{unpack}\ (X, x) = M; N]\!] & = & r \leftarrow [\![M]\!]; \mathtt{unpack}\ (X, f) = r; \\
 & & \mathtt{newcase}_X\ c_X; x \leftarrow (\mathtt{force}\ f)\ c_X; [\![N]\!] \\
[\![\Lambda^\nu X. M]\!] & = & \mathtt{ret}\ (\mathtt{thunk}\ (\Lambda X. \lambda (c_X : \mathrm{Case}\ X). [\![M]\!])) \\
[\![\Lambda^\nu \{X. ([A_n^{\sqsubseteq}\updownarrow, \ldots, A_1^{\sqsubseteq}\updownarrow], M)\}]\!] & = & \mathtt{ret}\ (\mathtt{thunk}\ (\Lambda X. \lambda (c_X : \mathrm{Case}\ X). M_n)) \\
 & & \text{where } M'_0 = [\![M]\!] \\
 & & M'_{i+1} = \\
 & & [\![A_{i+1}^{\sqsubseteq}]\!]\!\updownarrow_{i+1} [\mathtt{force}\ (\mathtt{thunk}\ (\Lambda X. . \lambda c_X : \mathrm{Case}\ X. [\![M_i]\!]))\ [X]\ c_X] \\
[\![\mathtt{let}\ x = M\{X \cong A\}; N]\!] & = & f \leftarrow [\![M]\!]; \mathtt{newcase}_{[\![A]\!]}\ c_X; \\
 & & x \leftarrow f\ [\![A]\!]\ c_X; [\![N]\!] \\
[\![\lambda (x : A). M]\!] & = & \mathtt{ret}\ \mathtt{thunk}\ \lambda (x : [\![A]\!]). [\![M]\!] \\
[\![M\ N]\!] & = & f \leftarrow [\![M]\!]; a \leftarrow [\![N]\!]; (\mathtt{force}\ f)\ a
\end{array}
$$

Figure 10.17: PolyC$^\nu$ term translation

$$
\begin{array}{lcl}
[\![G]\!]\updownarrow & = & \bullet \quad (\text{when } G = ?,\ \alpha, \text{ or } \texttt{Bool}) \\
[\![\text{tag}_G(A^{\sqsubseteq})]\!]\updownarrow & = & r \leftarrow [\![A^{\sqsubseteq}]\!]\updownarrow\,[\bullet]; \texttt{ret inj}_{\text{case}(G)}\ r \\
[\![\text{tag}_G(A^{\sqsubseteq})]\!]\downarrow & = & x \leftarrow \bullet; \texttt{match}\ x\ \texttt{with case}(G)\{\texttt{inj}\ y.[\![A^{\sqsubseteq}]\!]\downarrow[\texttt{ret}\ y] \mid \mho\} \\
[\![A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}]\!]\updownarrow & = & x \leftarrow \bullet; \texttt{let}\ (x_1, x_2) = x; \\
 & & x_1' \leftarrow [\![A_1^{\sqsubseteq}]\!]\updownarrow[\texttt{ret}\ x_1]; x_2' \leftarrow [\![A_2^{\sqsubseteq}]\!]\updownarrow[\texttt{ret}\ x_2]; \texttt{ret}\ (x_1', x_2') \\
[\![A_1^{\sqsubseteq} \to A_2^{\sqsubseteq}]\!]\updownarrow & = & x \leftarrow \bullet; \texttt{ret thunk}\ (\lambda y : A'.a \leftarrow [\![A_1^{\sqsubseteq}]\!]\updownarrow^{-}[\texttt{ret}\ y]; [\![A_2^{\sqsubseteq}]\!]\updownarrow[\texttt{force}\ x\ a]) \\
 & & \text{where } A_1^{\sqsubseteq} : A_{1l} \sqsubseteq A_{1r} \text{ and if } \updownarrow = \nwarrow, A' = A_{1r}, \text{ else } A' = A_{1l} \\
[\![\forall^{\nu} X.A^{\sqsubseteq}]\!]\updownarrow & = & x \leftarrow \bullet; \texttt{ret thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.[\![A^{\sqsubseteq}]\!]\updownarrow[\texttt{force}\ x\ [X]\ c_X]) \\
[\![\exists^{\nu} X.A^{\sqsubseteq}]\!]\updownarrow & = & x \leftarrow \bullet; \texttt{unpack}\ (Y, f) = x; \\
 & & \texttt{ret pack}(Y, \texttt{thunk}\ (\lambda c_X : \text{Case}\ Y.[\![A^{\sqsubseteq}]\!]\updownarrow[(\texttt{force}\ f)\ c_X]))
\end{array}
$$

Figure 10.18: PolyC$^{\nu}$ cast translation

Variables translate to a return of the variable, let is translated bind, and errors are translated to errors. Since the type translation maps known type variables to their bound types, the target language `seal` and `unseal` disappear in the translation. Injection into the dynamic type translates to injection into the open sum type and ground type checks in PolyC$^{\nu}$ are implemented using pattern matching on OSum in CBPV$_{\text{OSum}}$.

Next, we cover the cases involving thunks. As a warmup, the functions follow the usual CBV translation into CBPV: a CBV λ is translated to a thunk of a CBPV λ, and the application translation makes the evaluation order explicit and forces the function with an input. We translate existential packages in the cast calculus to CBPV$_{\text{OSum}}$ packages containing functions from a case of the open sum type to the body of the package. In PolyC$^{\nu}$ we delay execution of `pack` bodies, so the translation inserts a `thunk` to make the order of execution explicit. Since `pack` bodies translate to functions, the translation of an `unpack` must provide a case of the open sum type to the package it unwraps. Type abstractions (Λ^{ν}), like `pack`s, wrap their bodies in functions that, on instantiation, expect a case of the open sum type matching the instantiating type. Type application, similar to unpack, generates the requisite type tag using a `newcase` and passes its given type and that fresh tag associated with its type variable into the supplied type abstraction.

Next, we define the implementation of casts as "contracts", i.e., ordinary functions in the CBPV$_{\text{OSum}}$. Reflexive casts at atomic types, ?, α, and `Bool`, translate away. Structural casts at composite types, pair types, function types, universals, and existentials, push casts for their sub-parts into terms of each type. Function and product casts are entirely standard, noting that we use $r(A^{\sqsubseteq}) = A_r$. Universal casts delay until type application and then cast the output. Existential casts push their subcasts into whatever package they are given.

10.5.4 *Adequacy*

Next, to show that we can reason about PolyC$^{\nu}$ reductions by studying the corresponding CBPV$_{\text{OSum}}$ reductions of their translations, we need to prove *adequacy*, i.e., that programs diverge, error or terminate if and only if their translations do. We will show this by proving a *simulation* relation between source and target operational semantics. A naïve attempt at a simulation theorem would be to show that values and errors are preserved by the translation, and that if $M_1 \mapsto M_2$ then $[\![M_1]\!] \mapsto [\![M_2]\!]$. However, both of these notions fail. First, values are *not* directly preserved by the translation because the translation introduces *administrative redexes*. For instance $(\texttt{true}, \texttt{false})$ is translated to $\texttt{ret true} \leftarrow x; \texttt{ret false} \leftarrow y; \texttt{ret}\ (x, y)$. Note that this also means that some reductions in the source correspond to multiple reductions in the target, since the translation of $\texttt{let}\ (x, y) = (\texttt{true}, \texttt{false}); N$ will have to first reduce the scrutinee to a value before performing the pattern match. Second, some source reductions, such as $\texttt{unseal}_\sigma \texttt{seal}_\sigma V \mapsto V$ correspond to no steps in the target. This presents a challenge to showing that diverging terms are translated to diverging terms. Finally, because of administrative redexes, $[\![M_1]\!]$ may never reduce to $[\![M_2]\!]$ exactly. For instance in a β reduction $(\lambda x.M)V \mapsto M[V/x]$ the translation of $(\lambda x.M)V$ will reduce V's translation to a value and then substitute, whereas $M[V/x]$ will include administrative redexes everywhere x occurs in M, meaning they may never be eliminated because they occur under binders. To address these issues, we introduce two notions. First, to account for administrative redexes, we define a translation *relation* $M \rightsquigarrow^{CT} M'$ between PolyC$^{\nu}$ and CBPV$_{\text{OSum}}$ terms that we can think of as a multi-valued translation function that non-deterministically eliminates some administrative redexes. This translation relation generalizes the translation in that $M \rightsquigarrow^{CT} [\![M]\!]$, and because we can eliminate administrative redexes, we can also show that every value translates to some CBPV$_{\text{OSum}}$value: $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. Then we will be able to show that simulation is preserved Second, to account for invisible steps, we introduce a *size* function on source terms, $|\cdot|$, and show that whenever a PolyC$^{\nu}$ reduction does not correspond to a CBPV$_{\text{OSum}}$ step that the size of the PolyC$^{\nu}$ term reduces, so that a PolyC$^{\nu}$ term cannot reduce indefinitely without at least one corresponding CBPV$_{\text{OSum}}$ step.

The translation relation is defined in Figures 10.19 and 10.20. The first rule allows for the non-deterministic elimination of any *bind* redex in the translation. This includes all administrative redexes, though also some others. All of the remaining rules are just relational reformulations of the cases of the translation function. Since all of the original cases are included, it is very easy to see by induction that our translation function is included in our multi-function.

Lemma 194. *For any* $M \rightsquigarrow^{CT} [\![M]\!]$.

$$\frac{M \rightsquigarrow^{CT} x \leftarrow \mathtt{ret}\ V; N'}{M \rightsquigarrow^{CT} N'[V/x]} \qquad \frac{}{\mho \rightsquigarrow^{CT} \mho} \qquad x \rightsquigarrow^{CT} \mathtt{ret}\ x$$

$$\frac{M \rightsquigarrow^{CT} M' \qquad N \rightsquigarrow^{CT} N'}{\mathtt{let}\ x = M; N \rightsquigarrow^{CT} x \leftarrow M'; N'}$$

$$\frac{M_1 \rightsquigarrow^{CT} M_1' \qquad M_2 \rightsquigarrow^{CT} M_2' x}{(M_1, M_2) \rightsquigarrow^{CT} x_1 \leftarrow M_1'; x_2 \leftarrow M_2'; \mathtt{ret}\ (x_1, x_2)}$$

$$\frac{M \rightsquigarrow^{CT} M' \qquad N \rightsquigarrow^{CT} N'}{\mathtt{let}\ (x,y) = M; N \rightsquigarrow^{CT} r \leftarrow M'; \mathtt{let}\ (x,y) = r; N'}$$

$$\frac{}{\mathtt{true} \rightsquigarrow^{CT} \mathtt{ret}\ \mathtt{true}} \qquad \frac{}{\mathtt{false} \rightsquigarrow^{CT} \mathtt{ret}\ \mathtt{false}}$$

$$\frac{M \rightsquigarrow^{CT} M' \qquad M_1 \rightsquigarrow^{CT} M_1' \qquad M_2 \rightsquigarrow^{CT} M_2'}{\mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \rightsquigarrow^{CT} r \leftarrow M'; \mathtt{if}\ r\ \mathtt{then}\ M_1'\ \mathtt{else}\ M_2'}$$

$$\frac{M \rightsquigarrow^{CT} M' : A \to B \qquad N \rightsquigarrow^{CT} N' : A}{M\ N \rightsquigarrow^{CT} f \leftarrow M'; a \leftarrow N'; (\mathtt{force}\ f)\ a}$$

$$\frac{M \rightsquigarrow^{CT} M' : B}{\lambda x : A.M \rightsquigarrow^{CT} \mathtt{ret}\ \mathtt{thunk}\ \lambda x : [\![A]\!].M'} \qquad \frac{M \rightsquigarrow^{CT} M'}{\mathtt{seal}_X M \rightsquigarrow^{CT} M}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\mathtt{unseal}_X M \rightsquigarrow^{CT} M'}$$

Figure 10.19: PolyC$^{\nu}$ translation relation

$$\frac{M \rightsquigarrow^{CT} M'}{\langle A^{\sqsubseteq}\rangle\!\updownarrow\, M \rightsquigarrow^{CT} [\![A^{\sqsubseteq}]\!]\!\updownarrow\, [M']} \qquad \frac{M \rightsquigarrow^{CT} M'}{\langle A^{\sqsubseteq}\rangle\!\downarrow\, M \rightsquigarrow^{CT} [\![A^{\sqsubseteq}]\!]\!\downarrow\, [M']}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\begin{array}{l}\texttt{is}(G)\texttt{?}\ M \rightsquigarrow^{CT} \quad r \leftarrow M';\\ \qquad\qquad \texttt{match case}(G)\ \texttt{with}\ r\{\texttt{inj}\ y.\texttt{ret true} \mid \texttt{ret false}\}\end{array}}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\texttt{pack}^{\nu}(X \cong A, M) \rightsquigarrow^{CT} \texttt{ret pack}([\![A]\!], \texttt{thunk}\ \lambda(c_X : \text{Case}\ [\![A]\!]).M')\ \texttt{as}\ [\![\exists^{\nu} X.B]\!]}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\texttt{pack}^{\nu}(X \cong A, [], M) \rightsquigarrow^{CT} \texttt{ret pack}([\![A]\!], \texttt{thunk}\ \lambda(c_X : \text{Case}\ [\![A]\!]).M')\ \texttt{as}\ [\![\exists^{\nu} X.B]\!]}$$

$$\frac{\texttt{pack}^{\nu}(X \cong A, [\overline{A\ \updownarrow}], M) \rightsquigarrow^{CT} \texttt{ret pack}(A', \texttt{thunk}\ \lambda(c_X : \text{Case}\ A').M')}{\begin{array}{l}\texttt{pack}^{\nu}(X \cong A, [A_1\ \updownarrow_1, \overline{A\ \updownarrow}], M) \rightsquigarrow^{CT}\\ \quad \texttt{ret pack}(A', \quad \texttt{thunk}\ \lambda c_X : \text{Case}\ A'.\\ \qquad\qquad [\![A^{\sqsubseteq}]\!]\ \updownarrow_1\ [(\texttt{force thunk}\ \lambda c_X : \text{Case}\ A'.M')\ c_X])\end{array}}$$

$$\frac{M \rightsquigarrow^{CT} M' \qquad N \rightsquigarrow^{CT} N'}{\begin{array}{l}\texttt{unpack}\ (X, x) = M; N \rightsquigarrow^{CT} \quad r \leftarrow M'; \texttt{unpack}\ (X, f) = r;\\ \qquad\qquad\qquad\qquad\quad \texttt{newcase}_X\ c_X; x \leftarrow (\texttt{force}\ f)\ c_X; N'\end{array}}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\Lambda^{\nu} X.M \rightsquigarrow^{CT} \texttt{ret thunk}\ \Lambda X.\lambda(c_X : \text{Case}\ X).M'}$$

$$\frac{M \rightsquigarrow^{CT} M'}{\Lambda^{\nu}\{X.([], M)\} \rightsquigarrow^{CT} \texttt{ret thunk}\ \Lambda X.\lambda(c_X : \text{Case}\ X).M'}$$

$$\frac{\Lambda^{\nu}\{X.([\overline{B^{\sqsubseteq}\ \updownarrow}], M)\} \rightsquigarrow^{CT} \texttt{ret}\ V'}{\Lambda^{\nu}\{X.([B_1^{\sqsubseteq}\ \updownarrow_1, \overline{B^{\sqsubseteq}\ \updownarrow}], M)\} \rightsquigarrow^{CT} \texttt{ret thunk}\ \Lambda X.\lambda(c_X : \text{Case}\ X).[\![B_1^{\sqsubseteq}]\!]\ \updownarrow_1\ [(\texttt{force}\ V')\ [X]\ c_X]}$$

$$\frac{M \rightsquigarrow^{CT} M' \qquad N \rightsquigarrow^{CT} N'}{\texttt{let}\ x = M\{X \cong B\}; N \rightsquigarrow^{CT} f \leftarrow M'; \texttt{newcase}_{[\![B]\!]}\ c_X; x \leftarrow (\texttt{force}\ f)\ [\![B]\!]\ c_X; N'}$$

Figure 10.20: PolyC$^{\nu}$ translation relation (Continued)

Next, we prove a few simple lemmas about how the translation relation interacts with substitutions and evaluation contexts, since these are central to the operational semantics. First, we want a translation relation that directly relates cast calculus values (respectively evaluation contexts) to $\text{CBPV}_{\text{OSum}}$ values (respectively stacks). For values, we can re-use the term relation, but restricting the output to be return of a value: $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. For evaluation contexts, we will need a separate translation relation, defined in Figure 10.21. The relation is straightforward, essentially reducing the administrative redexes that would be present in the direct translation.

Next, we establish a few lemmas about these value and evaluation context translations. We need both *forward* reasoning, that says that it is a valid translation to homomorphically translate substitutions and evaluation contexts, but we also need *backwards* reasoning that says that any translation is essentially equivalent to a homomorphic translation.

First, we establish forward reasoning. We show that the relation is a congruence with respect to substituting values for variables, seals for types and terms in evaluation contexts.

Lemma 195. *If $M \rightsquigarrow^{CT} M'$ and $V_x \rightsquigarrow_v^{CT} V_x'$ then $M[V_x/x] \rightsquigarrow^{CT} M'[V_x'/x]$, and if $V \rightsquigarrow_v^{CT} V'$ and $V_x \rightsquigarrow_v^{CT} V_x'$ then $V[V_x/x] \rightsquigarrow_v^{CT} V'[V_x'/x]$. and if $E \rightsquigarrow_v^{CT} S'$ and $V_x \rightsquigarrow_v^{CT} V_x'$ then $E[V_x/x] \rightsquigarrow_v^{CT} S'[V_x'/x]$.*

Lemma 196. *If $M \rightsquigarrow^{CT} M'$ and X abstract in M and $\sigma : A$ then $M[\sigma/X] \rightsquigarrow^{CT} M'[[\![A]\!]/X][\sigma/c_X]$. Similarly for values and evaluation contexts.*

Lemma 197. *If $M \rightsquigarrow^{CT} M'$ $X \cong A$ in M and $\sigma : A$ then $M[\sigma/X] \rightsquigarrow^{CT} M'[\sigma/c_X]$. Similarly for values and evaluation contexts.*

Lemma 198. *If $M \rightsquigarrow^{CT} M'$ and $E \rightsquigarrow^{CT} S'$, then $E[M] \rightsquigarrow^{CT} M'[S']$.*

For *backwards* reasoning, it would be most convenient to know that if $E[M] \rightsquigarrow^{CT} N'$ then N' can be split into a pieces $S'[M']$ where $E \rightsquigarrow_s^{CT} S'$ and $M \rightsquigarrow^{CT} M'$, but this might not be the case because (1) N' may contain administrative redexes that will be evaluated before the stack is S' but also (2) if S is a bind, it might be reduced away and already be evaluated in N'. However these are really the only two obstacles in that N' must be equivalent to some $S'[M']$ except that each might have some binds that the other does not. Fortunately this will be good enough to prove our simulation relation. We define *bind reduction* to be the subset of the reduction relation that consists only of $F\beta$ reductions:

Definition 199. Define the relation $M \mapsto_b M'$ inductively by the single rule:

$$S[x \leftarrow \texttt{ret}\ V; N] \mapsto_b S[N[V/x]]$$

Define $\mapsto_b^*$ to be the transitive closure of $\mapsto_b$.
Define $\cong_b$ to be the induced equivalence relation

$$M \cong_b M' = \exists N.\ M \mapsto_b^* N \wedge M' \mapsto_b^* N$$

$$[\cdot] \rightsquigarrow_s^{CT} \bullet \qquad \frac{E \rightsquigarrow_s^{CT} S' \qquad N \rightsquigarrow^{CT} N'}{\texttt{let}\ x = E; N \rightsquigarrow_s^{CT} x \leftarrow S'; N'}$$

$$\frac{E \rightsquigarrow_s^{CT} S' \qquad M \rightsquigarrow^{CT} M'}{(E, M) \rightsquigarrow_s^{CT} x \leftarrow S'; y \leftarrow M'; \texttt{ret}\ (x, y)}$$

$$\frac{V \rightsquigarrow^{CT} \texttt{ret}\ V' \qquad E \rightsquigarrow_s^{CT} S'}{(V, E) \rightsquigarrow_s^{CT} y \leftarrow S'; \texttt{ret}\ (V', y)}$$

$$\frac{E \rightsquigarrow_s^{CT} S' \qquad M \rightsquigarrow^{CT} M'}{E\ M \rightsquigarrow_s^{CT} f \leftarrow S'; x \leftarrow M'; (\texttt{force}\ f)\ x}$$

$$\frac{V \rightsquigarrow^{CT} \texttt{ret}\ V' \qquad E \rightsquigarrow_s^{CT} S'}{V\ E \rightsquigarrow_s^{CT} x \leftarrow S'; (\texttt{force}\ V')\ x}$$

$$\frac{E \rightsquigarrow_s^{CT} E' \qquad M \rightsquigarrow^{CT} M' \qquad N \rightsquigarrow^{CT} N'}{\texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ N \rightsquigarrow_s^{CT} x \leftarrow S'; \texttt{if}\ x\ \texttt{then}\ M'\ \texttt{else}\ N'}$$

$$\frac{E \rightsquigarrow_s^{CT} E' \qquad M \rightsquigarrow^{CT} M'}{\texttt{let}\ (x, y) = E; M \rightsquigarrow_s^{CT} p \leftarrow S'; \texttt{let}\ (x, y) = p; M'}$$

$$\frac{E \rightsquigarrow_s^{CT} S' \qquad N \rightsquigarrow^{CT} N'}{\texttt{unpack}\ (X, x) = E; N \rightsquigarrow_s^{CT} \left(\begin{array}{l} r \leftarrow S'; \\ \texttt{unpack}\ (X, f) = r; \\ \texttt{newcase}_X\ c_X; \\ x \leftarrow (\texttt{force}\ f)\ c_X; \\ N' \end{array}\right)}$$

$$\frac{E \rightsquigarrow_s^{CT} S' \qquad N \rightsquigarrow^{CT} N'}{\texttt{let}\ x = E\{X \cong B\}; N \rightsquigarrow_s^{CT} f \leftarrow S'; \texttt{newcase}_{[\![B]\!]}\ c_X; x \leftarrow (\texttt{force}\ f)\ [\![B]\!]\ c_X; N'}$$

$$\frac{E \rightsquigarrow_s^{CT} S'}{\texttt{seal}_X E \rightsquigarrow_s^{CT} S} \qquad \frac{E \rightsquigarrow_s^{CT} S'}{\texttt{unseal}_X E \rightsquigarrow_s^{CT} S'} \qquad \frac{E \rightsquigarrow_s^{CT} S'}{\langle A^{\sqsubseteq} \rangle\!\updownarrow\ E \rightsquigarrow_s^{CT} [\![A^{\sqsubseteq}]\!]\!\updownarrow [S']}$$

$$\frac{E \rightsquigarrow_s^{CT} S'}{\texttt{is}(G)\texttt{?}\ E \rightsquigarrow_s^{CT} \left(\begin{array}{l} r \leftarrow S'; \\ \texttt{match case}(G)\ \texttt{with}\ r\{\texttt{inj}\ y.\texttt{ret true} \mid \texttt{ret false}\} \end{array}\right)}$$

Figure 10.21: PolyC$^{\nu}$ translation relation, evaluation contexts

To justify that $\cong_b$ really is an equivalence relation, we prove that bind reduction is normalizing and therefore confluent.

Lemma 200 (bind reduction confluence)**.** *Let* $\Sigma;\cdot \vdash M_1 : A$, $\Sigma;\cdot \vdash M_2 : A$, *and* $\Sigma;\cdot \vdash M_3 : A$. *If* $M_1 \mapsto_b{}^* M_2$, $M_1 \mapsto_b{}^* M_3$, *and* M_3 *does not take a bind reduction, then* $M_2 \mapsto_b{}^* M_3$.

Proof. By induction on $M_1 \mapsto_b{}^* M_2$. If $M_1 = M_2$, then we conclude since $M_1 \mapsto_b{}^* M_3$. Otherwise, by the definition of $\mapsto_b{}^*$, we have $M_1 \mapsto_b N_1 \mapsto_b{}^* M_2$. Since M_1 takes a bind reduction, $M_1 \neq M_3$ and so we must have $M_1 \mapsto_b N_2 \mapsto_b{}^* M_3$. Furthermore, by Lemma 189 (deterministic semantics), we have $N_1 = N_2$. We then conclude by the inductive hypothesis for N_2, M_2, M_3. □

And normalizing,

Lemma 201 (bind reduction normalization)**.** *Let* $\Sigma;\cdot \vdash M_1 : A$. *There exists a unique* M_2 *such that* $M_1 \mapsto_b{}^* M_2$ *and* M_2 *does not take a bind reduction.*

Proof. By induction on the number of binds in M_1 not under thunks. If $M_1 = S[x \leftarrow \texttt{ret}\ V; N_1]$ then $M_1 \mapsto_b N_1[V/x]$ and we conclude by the inductive hypothesis for $N_1[V/x]$. Otherwise, M_1 must not take a bind reduction, so $M_2 = M_1$ and we conclude by reflexivity. □

Lemma 202. *If* $M_1 \cong_b M_2$, *and* $M_1 \mapsto^+ N$ *where at least one of the steps is* not *a bind reduction, then* $M_2 \mapsto^+ N$.

Proof. First, there is some M_c with $M_1 \mapsto_b^* M_c$ and $M_2 \mapsto_b^* M_c$.

Since reduction is deterministic, the sequence must factorize as $M_1 \mapsto_b^* M_c \mapsto_b^* M_1' \mapsto_{nb} \mapsto^* N$. So $M_2 \mapsto_b^* M_c \mapsto^+ N$ □

Lemma 203 (bind equivalence transitivity)**.** *If* $M_1 \cong_b M_2$ *and* $M_2 \cong_b M_3$ *then* $M_1 \cong_b M_3$.

Proof. M_1 and M_2 both bind-step to N_1 and M_2 and M_3 both bind-step to N_2. By confluence there is some N_3 that N_1, N_2 both step to, and so $M_1 \cong_b M_3$. □

Then we can show that, reasoning backwards, any translation of a term is "bind-equivalent" to a homomorphic translation. First, any translation of a value bind-reduces to a value translation.

Lemma 204. *If* $V \rightsquigarrow^{CT} M'$ *then there exists* V' *with* $V \rightsquigarrow_v^{CT} V'$ *such that* $M' \cong_b \texttt{ret}\ V'$

Proof. By induction on V. All cases are immediate since $M' = \texttt{ret}\ V'$ except the following:

- If $V = \texttt{seal}_\alpha V$ or $V = \langle \text{tag}_G(G) \rangle \updownarrow V_2$, then it follows by inductive hypothesis.

- $(V_1, V_2) \rightsquigarrow^{CT} x_1 \leftarrow N_1'; x_2 \leftarrow N_2'; \mathtt{ret}\ (x_1, x_2)$ where $V_1 \rightsquigarrow^{CT} N_1'$ and $V_2 \rightsquigarrow^{CT} N_2'$: By the inductive hypothesis for each of these sub-derivations, we have $N_i' \mapsto_b^* \mathtt{ret}\ V_i'$ and $V_i \rightsquigarrow^{CT} \mathtt{ret}\ V_i'$. Then, by definition, we have $M' \mapsto_b^* \mathtt{ret}\ (V_1, V_2)$ and $(V_1, V_2) \rightsquigarrow^{CT} \mathtt{ret}\ (V_1, V_2)$ as we were required to show.

□

Next, we need a similar lemma for evaluation contexts, that says that if we translate a term in an evaluation context, then the result is bind-equivalent to something that can be decomposed into a homomorphic translation. Unlike the value case, we need to allow the general bind-equality here. This requires a simple lemma that says bind-equivalence is a congruence with respect to stack plugging.

Lemma 205 (plug is a congruence). *Let $\Sigma; \cdot \vdash M_1 : A$ and $\Sigma; \cdot \mid A \vdash S : B$. For any closing γ, if $M_1[\gamma] \cong_b M_2[\gamma]$, then $S[M_1][\gamma] \cong_b S[M_2][\gamma]$.*

Proof. Let N be the witness to $M_1[\gamma] \cong_b M_2[\gamma]$. Then $S[M_1][\gamma] \mapsto_b^* S[N][\gamma]$ and $S[M_2][\gamma] \mapsto_b^* S[N][\gamma]$, so $S[N][\gamma]$ witnesses $S[M_2][\gamma] \cong_b S[M_2][\gamma]$. □

Lemma 206. *If $E[N] \rightsquigarrow^{CT} M'$, then there exists S', N' with $E \rightsquigarrow_s^{CT} S'$ and $N \rightsquigarrow^{CT} N'$ such that $M' \cong_b S'[N']$.*

Proof. By induction on the derivation of $E[M_1] \rightsquigarrow^{CT} M'$. We prove the critical cases below:

- $$\frac{M \rightsquigarrow^{CT} x \leftarrow \mathtt{ret}\ V_1'; N}{M \rightsquigarrow^{CT} N[V_1'/x]}$$

  From the inductive hypothesis, we have some S, M_1' such that $S[M_1'][\gamma] \cong_b (x \leftarrow \mathtt{ret}\ V_1'; N)[\gamma]$, $M_1 \rightsquigarrow^{CT} M_1'$, and $E \rightsquigarrow^{CT} S$. Since bind reduction preserves bind equivalence, we have $(x \leftarrow \mathtt{ret}\ V_1'; N)[\gamma] \cong_b N[V_1'/x]$ and so we conclude by transitivity (Lemma 203).

- $$\frac{M_3 \rightsquigarrow^{CT} M_3' \qquad M_2 \rightsquigarrow^{CT} M_2'}{(M_3, M_2) \rightsquigarrow^{CT} x_3 \leftarrow M_3'; x_2 \leftarrow M_2'; \mathtt{ret}\ (x_3, x_2)}$$

  We have two cases based on whether the context hole lies on the left or right side of the pair.

  1. Consider the case where $E = (E_3, M_2)$. Then we have some S_3, M_1' such that $S_3[M_1'][\gamma] \cong_b M_3'[\gamma]$, $M_1 \rightsquigarrow^{CT} M_1'$, and $E_3 \rightsquigarrow^{CT} S_3$. Let $S_p = x_3 \leftarrow \bullet; x_2 \leftarrow M_2'; \mathtt{ret}\ (x_3, x_2)$ and let $S = S_p[S_3]$. Then $E \rightsquigarrow^{CT} S$ by definition since $E_3 \rightsquigarrow^{CT} S_3$. It remains to show $S[M_1'][\gamma] \cong_b M'[\gamma]$. We have $M' = S_p[M_3']$, so we conclude by Lemma 205 (plug is a congruence) since $S_3[M_1'][\gamma] \cong_b M_3'[\gamma]$.

2. Consider the case where $E = (V_3, E_2)$. Then by Lemma 204, $M_3'[\gamma] \mapsto_b^* \texttt{ret}\ V_3'[\gamma]$ for some V_3' such that $M_3 \rightsquigarrow^{CT} \texttt{ret}\ V_3'$ and we have some S_2, M_1' such that $S_2[M_1'][\gamma] \cong_b M_2'[\gamma]$, $M_1 \rightsquigarrow^{CT} M_1'$, and $E_2 \rightsquigarrow^{CT} S_2$. Let $S_p = x_2 \leftarrow \bullet; \texttt{ret}\ (V_3', x_2)$ and let $S = S_p[S_2]$. Then $E \rightsquigarrow^{CT} S$ by definition, since $E_2 \rightsquigarrow^{CT} S_2$ and $M_3 \rightsquigarrow^{CT} \texttt{ret}\ V_3'$, so it remains to show $S[M_1'][\gamma] \cong_b M'[\gamma]$. Note that $M'[\gamma] \mapsto_b^* S_p[M_2']$, so we conclude by Lemmas 203 (transitivity) and 205 (plug is a congruence).

- $$\frac{M_3 \rightsquigarrow^{CT} M_3' \qquad M_2 \rightsquigarrow^{CT} M_2'}{M_3\ M_2 \rightsquigarrow^{CT} f \leftarrow M_3'; a \leftarrow M_2'; (\texttt{force}\ f)\ a}$$

  We have two cases based on whether the context hole lies on the left or right side of the application.

  1. Consider the case where $E = E_3\ M_2$. Then we have some S_3, M_1' such that $S_3[M_1'][\gamma] \cong_b M_3'[\gamma]$, $M_1 \rightsquigarrow^{CT} M_1'$, and $E_3 \rightsquigarrow^{CT} S_3$. Let $S_a = f \leftarrow \bullet; a \leftarrow M_2'; (\texttt{force}\ f)\ a$ and let $S = S_a[S_3]$. Then $E \rightsquigarrow^{CT} S$ by definition since $E_3 \rightsquigarrow^{CT} S_3$. It remains to show $S[M_1'][\gamma] \cong_b M'[\gamma]$. We have $M' = S_a[M_3']$, so we conclude by Lemma 205 (plug is a congruence) since $S_3[M_1'][\gamma] \cong_b M_3'[\gamma]$.
  2. Consider the case where $E = V_3\ E_2$. Then by Lemma 204, $M_3'[\gamma] \mapsto_b^* \texttt{ret}\ V_3'[\gamma]$ for some V_3' such that $M_3 \rightsquigarrow^{CT} \texttt{ret}\ V_3'$ and we have some S_2, M_1' such that $S_2[M_1'][\gamma] \cong_b M_2'[\gamma]$, $M_1 \rightsquigarrow^{CT} M_1'$, and $E_2 \rightsquigarrow^{CT} S_2$. Let $S_a = a \leftarrow M_2'; (\texttt{force}\ V_3')\ a$ and let $S = S_a[S_2]$. Then $E \rightsquigarrow^{CT} S$ by definition since $E_2 \rightsquigarrow^{CT} S_2$ and $M_3 \rightsquigarrow^{CT} \texttt{ret}\ V_3'$, so it remains to show $S[M_1'][\gamma] \cong_b M'[\gamma]$. Note that $M'[\gamma] \mapsto_b S_a[M_2']$, so we conclude by Lemmas 203 (transitivity) and 205 (plug is a congruence).

All other cases are analogous to the first part of the pair and function application cases. □

Next, clearly the translation multi-function is closed under $\mapsto_b$ reductions.

Lemma 207 (Bind reduction preserves translation)**.** *If $M \rightsquigarrow^{CT} M_1'$ and $M_1' \mapsto_b M_2'$, then $M \rightsquigarrow^{CT} M_2'$.*

Next, we need a simple lemma that tells us that we can reason about reduction of a term by reasoning about a term that is bind-equivalent.

Lemma 208. *If $M_1 \cong_b M_2$, and $M_1 \mapsto^+ N$ where at least one of the steps is* not *a bind reduction, then $M_2 \mapsto^+ N$.*

Proof. First, there is some M_c with $M_1 \mapsto_b^* M_c$ and $M_2 \mapsto_b^* M_c$.

Since reduction is deterministic, the sequence must factorize as $M_1 \mapsto_b^* M_c \mapsto_b^* M_1' \mapsto_{nb} \mapsto^* N$. So $M_2 \mapsto_b^* M_c \mapsto^+ N$ □

To simplify backwards reasoning, note that since the only non-determinism is in reduction of bind-redexes, every translation is bind-equivalent to one that starts with a translation, rather than reduction rule.

Lemma 209 (canonical forms of the translation relation)**.** *If* $\Sigma;\cdot \vdash M \rightsquigarrow^{CT} M' : B$*, then, there exists* M'_c *such that* $\Sigma;\cdot \vdash M \rightsquigarrow^{CT} M'_c$ *where the first rule of* $M \rightsquigarrow^{CT} M'_c$ *is not a bind reduction and* $M'_c \mapsto_b^* M'$

We would like all PolyC$^\nu$ terms to make progress whenever their corresponding CBPV translation evaluates. However, this is not always the case since some PolyC$^\nu$ terms step to terms with identical translations. However, the number of such steps is limited by the syntactic size of the term, defined below. The difficulty in defining term size is that one of the potentially invisible reductions is *bind* reductions, so we need to ensure that $\texttt{let}\ x = V; N$ is a larger term by our metric than $N[V/x]$ even though syntactically N may contain many uses of x. To accommodate this, we *define* the size of a let to be one more than the size of the continuation with the discriminee substituted in $|N[M/x]|$. However this definition is not structurally recursive, so to do this we instead define size for *open* terms to be parameterized by a size for each of their free variables.

Definition 210 (Type/Term size)**.**

$$\begin{array}{lcl}
|\texttt{let}\ x = M; N|\gamma & = & 1 + |N|(\gamma, x \mapsto |M|\gamma) \\
|\lambda x.M|\gamma & = & 0 \\
|C(\vec{A^{\sqsubseteq}}, \vec{M})|\gamma & = & 1 + \sum_{M \in \vec{M}} |M|\gamma + \sum_{A^{\sqsubseteq} \in \vec{A^{\sqsubseteq}}} |A^{\sqsubseteq}| \text{ otherwise} \\
|\mathrm{tag}_G(A^{\sqsubseteq})| & = & 2 + |A^{\sqsubseteq}|
\end{array}$$

For closed terms, we then define $|M| = |M|\emptyset$

Note that term size interacts nicely with substitution in that

Lemma 211. $|M[V/x]|\gamma = |M|(\gamma, x \mapsto |V|)$

Proof. By induction on M. □

Our simulation theorem divides into two cases, either a source step is matched by a target step, or it corresponds to o steps in the target, but our size metric decreases. Since our size metric cannot decrease forever, we will be able to show that diverging source terms translate to diverging target terms.

Theorem 212 (Simulation)**.** *For any well-typed* $M_1 \rightsquigarrow^{CT} M'_1$*, if* $M_1 \mapsto M_2$*, then there exists* M'_2 *such that* $M_2 \rightsquigarrow^{CT} M'_2$ *and* $M'_1[\gamma_p] \mapsto^* M'_2[\gamma_p]$ *and either*

- $M'_1[\gamma_p] \mapsto^+ M'_2[\gamma_p]$
- $|M_2| < |M_1|$

Proof. In each case we have $M_1 = E[M_r]$ where M_r is either an error or a redex. By Lemma 206 (translation context decomposition) and Lemma 209, we have $M_1 \rightsquigarrow^{CT} S'[M_r'] \cong_b M_1'$ where $E \rightsquigarrow_s^{CT} S'$ and $M_r \rightsquigarrow^{CT} M_r'$ starting with a congruence rule.

- Error:

  $$E[\mho_B] \mapsto \mho_{type(\Sigma;\cdot \vdash E[\mho_B])} \text{ where } E \neq []$$

  $M_r' = \mho$ and $S[\mho][\gamma_p] \mapsto \mho$. So the result follows because $\mho \rightsquigarrow^{CT} \mho$

- Pack:

  $$E[\mathsf{pack}^\nu(X \cong A_1, N_1)] \mapsto E[\mathsf{pack}^\nu(X \cong A_1, [], N_1)]$$

  Note that $\mathsf{pack}^\nu(X \cong A_1, N_1) \rightsquigarrow^{CT} M_3'$ iff $\mathsf{pack}^\nu(X \cong A_1, [], N_1) \rightsquigarrow^{CT} M_3'$, so by Lemma 198 (translation context plug), we may choose $M_2' = M_r'$. Since there is no target step, we must show that $|M_1| > |M_2|$. This holds since $|\mathsf{pack}^\nu(X \cong A_1, N_1)| = 2 + |N_1| > 1 + |N_1| = |\mathsf{pack}^\nu(X \cong A_1, [], N_1)|$.

- Unpack:

$$\begin{array}{rll} & \Sigma \triangleright & E[\mathsf{unpack}\ (X, x) = \mathsf{pack}^\nu(X \cong A_1, [A^{\sqsubseteq} \updownarrow ...], N_1); N_2] \\ \mapsto & \Sigma, \sigma : \text{Case } A_1 \triangleright & E[\mathsf{let}\ x = \langle A^{\sqsubseteq}[\sigma/X]\rangle \updownarrow ...\ N_1[\sigma/X]; N_2[\sigma/X]] \end{array}$$

  Let $A' = [\![\Sigma; \cdot \vdash \exists^\nu X.B]\!]$ and $A_1' = [\![\Sigma; \cdot \vdash A_1]\!]$. Then there is a term

$$\begin{array}{rl} M_r'[\gamma_p] = & r \leftarrow \mathsf{ret}\ \mathsf{pack}(A_1', \mathsf{thunk}\ \lambda(c_X : \text{Case } A_1').N_{cst}'[\gamma_p])\ \mathsf{as}\ A'; \\ & \mathsf{unpack}\ (X, f) = r; \\ & \mathsf{newcase}_X\ c_X; \\ & x \leftarrow (\mathsf{force}\ f)\ c_X; \end{array}$$

with $N'_{cst} = [\![A^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force}\ (\texttt{thunk}\ (\lambda c_X : \text{Case}\ A'.(\ldots\ N'_1\ \ldots)))\ c_X]$ and $N_i \rightsquigarrow^{CT} N'_i$ such that $M'_1 \cong_b S'[M'_r]$ such that $E \rightsquigarrow^{CT}_s S'$. Reducing:

$$
\begin{array}{rll}
 & & \texttt{unpack}\ (X, f) = \texttt{pack}(A'_1, \texttt{thunk}\ \lambda(c_X : \text{Case}\ A'_1).N'_{cst}[\gamma_p])\ \texttt{as}\ A'; \\
 & & \texttt{newcase}_X\ c_X; \\
 & & x \leftarrow (\texttt{force}\ f)\ c_X; \\
 & & N'_2[\gamma_p] \\
\mapsto & [\![\Sigma]\!]\triangleright & \texttt{newcase}_{A'_1}\ c_X; \\
 & & x \leftarrow (\texttt{force thunk}\ \lambda(c_X : \text{Case}\ A'_1).N'_{cst}[\gamma_p])\ c_X; \\
 & & N'_2[\gamma_p][A'_1/X][\texttt{thunk}\ \lambda(c_X : \text{Case}\ A'_1).N'_{cst}[\gamma_p]/f] \\
= & [\![\Sigma]\!]\triangleright & \texttt{newcase}_{A'_1}\ c_X; \\
 & & x \leftarrow (\texttt{force thunk}\ \lambda(c_X : \text{Case}\ A'_1).N'_{cst}[\gamma_p])\ c_X; \\
 & & N'_2[\gamma_p][A'_1/X] \\
\mapsto^* & [\![\Sigma]\!], \sigma : A'_1\triangleright & x \leftarrow N'_{cst}[\gamma_p][\sigma/c_X]; \\
 & & N'_2[\gamma_p][A'_1/X][\sigma/c_X] \\
\mapsto^* & [\![\Sigma]\!], \sigma : A'_1\triangleright & x \leftarrow [\![A^{\sqsubseteq}]\!]^{\updownarrow}[\ldots\ N'_1\ \ldots][\gamma_p][\sigma/c_X]; \\
 & & N'_2[\gamma_p][A'_1/X][\sigma/c_X]
\end{array}
$$

Note that, since $\Sigma; X \cong A'_1 \vdash N_1 \rightsquigarrow^{CT} N'_1$, we have $\Sigma; X \cong A'_1 \vdash \langle A^{\sqsubseteq}[\sigma/X]\rangle^{\updownarrow}\ \ldots\ N_1 \rightsquigarrow^{CT} [\![A^{\sqsubseteq}]\!]^{\updownarrow}\ [\ldots\ N'_1\ \ldots]$ by the definition of the translation relation for casts. Furthermore, by Lemma 197, since $\sigma : A_1 \in \Sigma$, we have $\langle A^{\sqsubseteq}[\sigma/X]\rangle^{\updownarrow}\ \ldots\ N_1[\sigma/X] \rightsquigarrow^{CT} [\![A^{\sqsubseteq}]\!]^{\updownarrow}\ [\ldots\ N'_1\ \ldots][\sigma/c_X]$ and by Lemma 196, since $\sigma : A_1 \in \Sigma$ and $\Sigma; X \vdash N_2 \rightsquigarrow^{CT} N'_2 : A_2$, we have $N_2[\sigma/X] \rightsquigarrow^{CT} N'_2[A'_1/X][\sigma/c_X]$. Thus, we have

$$
\begin{array}{l}
\texttt{let}\ x = \langle A^{\sqsubseteq}[\sigma/X]\rangle^{\updownarrow}\ \ldots\ N_1[\sigma/X]; N_2[\sigma/X] \\
\rightsquigarrow^{CT} x \leftarrow [\![A^{\sqsubseteq}]\!]^{\updownarrow}[\ldots\ N'_1\ \ldots][\sigma/c_X]; N'_2[A'_1/X][\sigma/c_X]
\end{array}
$$

so we conclude by Lemma 198 (translation context plug).

- Universal Instantiation:

$$\Sigma\triangleright E[\texttt{let}\ x = (\{\Lambda^{\nu} \cong .X\}; \overline{A^{\sqsubseteq}\ \updownarrow} M_2) Y B N] \mapsto E[\texttt{let}\ N[\sigma/Y] = \overline{\langle A^{\sqsubseteq}[\sigma/X]\rangle^{\updownarrow}} M[\sigma/X];]$$

Let $B' = [\![\Sigma; \cdot \vdash B]\!]$. Let $V_f = \Lambda^{\nu} X.\overline{A^{\sqsubseteq}\ \updownarrow}$. We have

$$E[M_r] \rightsquigarrow^{CT} S'[(f \leftarrow M'_f; \texttt{newcase}_{B'}\ c_X; x \leftarrow (\texttt{force}\ f)\ [B']\ c_x; N')][\gamma_p] \cong_b M'_1[\gamma_p]$$

where $V_f \rightsquigarrow^{CT} M'_f$, $E \rightsquigarrow^{CT}_s S'$ and $N \rightsquigarrow^{CT} N'$. Next, we know $V_f \rightsquigarrow^{CT} \texttt{ret}\ V'_f$ and $M'_f \mapsto^*_b V'_f$, so

$$\Sigma\triangleright M'_1[\gamma_p] \cong_b \mapsto^+ \Sigma, \sigma : B'\triangleright x \leftarrow (\texttt{force}\ V'_f[\gamma_p])\ [B']\ c_x; N'[\gamma_p]$$

By plugging lemma, and the definition of the translation it is sufficient to show that

$$(\texttt{force}\ V'_f[\gamma_p])\ [B']\ c_x \mapsto^* \overline{[\![A^{\sqsubseteq}\ \updownarrow]\!]} M'_2$$

for some $M_2 \rightsquigarrow^{CT} M'_2$. This follows by induction on the list $\overline{[\![A^{\sqsubseteq}\ \updownarrow]\!]}$.

- function application

$$E[(\lambda x : A.N_1)\ V] \mapsto E[N_1[V/x]]$$

We have $S'[(f \leftarrow \texttt{ret thunk}\ \lambda x : [\![\Sigma;\cdot \vdash A]\!].N'_1; a \leftarrow N'_2; \texttt{force}\ f\ a)][\gamma_p] \cong_b M'_1$ where $N_i \rightsquigarrow^{CT} N'_i, E \rightsquigarrow^{CT}_s S'$. By Lemma 204 (value translation), we have some V' such that $N'_2[\gamma_p] \mapsto_b^*\ \texttt{ret}\ V'[\gamma_p]$ and $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. Reducing and using Lemma 208, we see $M'_1[\gamma_p] \mapsto^+ S'[N'_1[V/x]][\gamma_p]$.

And the conclusion follows by substitution and plugging lemmas.

- If true:

$$E[\texttt{if true then}\ N_1\ \texttt{else}\ N_2] \mapsto E[N_1]$$

We have $S[(r \leftarrow \texttt{ret true}; \texttt{if}\ r\ \texttt{then}\ N'_1\ \texttt{else}\ N'_2)][\gamma_p] \cong_b M'_1[\gamma_p]$ where $N_i \rightsquigarrow^{CT} N'_i$, $E \rightsquigarrow^{CT}_s S'$. Then by Lemma 208, and reducing we see $M'_1[\gamma_p] \mapsto^+ S'[N'_1][\gamma_p]$ and the result follows by the context plugging lemma

- If false: analogous to previous.

- Pair elimination:

$$\texttt{let}\ (x,y) = (V_1, V_2); N \mapsto N[V_1/x][V_2/y]$$

We have $M'_1[\gamma_p] \cong_b (r \leftarrow M''; \texttt{let}\ (x,y) = r; N')[\gamma_p]$ where $(V_1, V_2) \rightsquigarrow^{CT} M''$ and $N \rightsquigarrow^{CT} N', E \rightsquigarrow^{CT}_s S'$. Thus, by Lemma 204 (value translation), we further have $M''[\gamma_p] \mapsto_b^*\ \texttt{ret}\ V'[\gamma_p]$ and $(V_1, V_2) \rightsquigarrow^{CT} \texttt{ret}\ V'$ for some V'. Note that by the type of V', we have $V' = (V'_1, V'_2)$ for some V'_1, V'_2.

We then have by Lemma 208 and the operational semantics that $M'_1[\gamma_p] \mapsto^+ S[N'[V'_1/x][V'_2/y]][\gamma_p]$ and the result follows by substitution and plugging lemmas.

- Let:

$$\texttt{let}\ x = V; N_2 \mapsto N_2[V/x]$$

We have $M'_1[\gamma_p] \cong_b S'[(x \leftarrow N'_1; N'_2)][\gamma_p]$ where $V \rightsquigarrow^{CT} N'_1$ and $N_2 \rightsquigarrow^{CT} N'_2$ and $E \rightsquigarrow^{CT}_s S'$. Then, by Lemma 204 (value translation reduction), we have $N'_1[\gamma_p] \mapsto_b^*\ \texttt{ret}\ V'[\gamma_p]$ for some V' such that $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. Thus, we have the following reduction:

$$S[x \leftarrow N'_1; N'_2][\gamma_p] \mapsto_b^* S[N'_2[V'/x]][\gamma_p]$$

Note that now we know by substitution and plugging lemmas that $E[N_2[V/x]] \rightsquigarrow^{CT} S[N_2'[V'/x]]$, but we only know that $M'[\gamma_p] \cong_b S[N_2'[V'/x]]$.

By Lemma 201 (bind normalization) there exists a unique M_4' such that

$$S'[x \leftarrow N_1'; N_2'][\gamma_p] \mapsto_b^* M_4'[\gamma_p]$$

and $M_4'[\gamma_p]$ does not take a bind reduction. Then, by Lemma 200 (bind reduction confluence), we have $M_1'[\gamma_p] \mapsto_b^* M_4'[\gamma_p]$ and $S[N_2'[V'/x]][\gamma_p] \mapsto_b^* M_4'[\gamma_p]$.

Then, by Lemma 207 (bind reduction preserves translation), we have $E[N_2[V/x]] \rightsquigarrow^{CT} M_4'$.

It finally suffices to show that $|M_1| > |M_2|$, which we have since by definition $|\texttt{let}\ x = V; N_2| = 1 + |N_2[V/x]| > |N_2[V/x]|$.

- Unseal:

  $$E[\texttt{unseal}_\sigma \texttt{seal}_\sigma V] \mapsto E[V]$$

  By definition of the translation, $E[V] \rightsquigarrow^{CT} M_1'$ as well so it suffices to show that $|M_1| > |M_2|$. This holds since by definition $|M_3| = 1 + 1 + |V|$.

- Identity cast:

  $$E[\langle A^{\sqsubseteq} \rangle\!\updownarrow V] \mapsto E[V] \text{ where } A \in \{\texttt{Bool}, \alpha, ?\}$$

  The reasoning for this case is analogous to the prior case.

- Pair cast:

  $$E[\langle A_1^{\sqsubseteq} \times A_2^{\sqsubseteq} \rangle\!\updownarrow (V_1, V_2)] \mapsto E[(\langle A_1^{\sqsubseteq} \rangle\!\updownarrow V_1, \langle A_2^{\sqsubseteq} \rangle\!\updownarrow V_2)]$$

  We have $M_1'[\gamma_p] \cong_b S'[[\![A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}]\!]\!\updownarrow [x_1 \leftarrow N_1'; x_2 \leftarrow N_2'; \texttt{ret}\ (x_1, x_2)]][\gamma_p]$ where $V_i \rightsquigarrow^{CT} N_i'$, $E \rightsquigarrow_s^{CT} S'$. Then, by Lemma 204 (value translation reduction), we have $N_i'[\gamma_p] \mapsto_b^* \texttt{ret}\ V_i'[\gamma_p]$ for some V_1', V_2' such that $V_i \rightsquigarrow^{CT} \texttt{ret}\ V_i'$. Thus, reducing and using Lemma 208, we have

  $$\begin{aligned} M_1'[\gamma_p] \\ &\mapsto^+ S'[x_1' \leftarrow [\![A_1^{\sqsubseteq}]\!]\!\updownarrow [\texttt{ret}\ x_1][\gamma_p]; \\ &\quad x_2' \leftarrow [\![A_2^{\sqsubseteq}]\!]\!\updownarrow [\texttt{ret}\ x_2][\gamma_p]; \\ &\quad \texttt{ret}\ (x_1', x_2')] \end{aligned}$$

  And the result follows by context plugging and definition of the simulation relation.

- Function cast:

$$E[(\langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq}\rangle\updownarrow\, V_1)\ V_2] \mapsto E[\langle A_2^{\sqsubseteq}\rangle\updownarrow\, (V_1\ \langle A_1^{\sqsubseteq}\rangle\updownarrow^{-}\ V_2)]$$

  We have $M_1'[\gamma_p] \cong_b S[(f \leftarrow [\![A_1^{\sqsubseteq} \to A_2^{\sqsubseteq}]\!]\updownarrow[N_1'];a \leftarrow N_2';\texttt{force}\ f\ a)][\gamma_p]$ where $V_i \rightsquigarrow^{CT} N_i'$, $E \rightsquigarrow_s^{CT} S'$. Then, by Lemma 204 (value translation reduction), we have $N_i'[\gamma_p] \mapsto_b^* \texttt{ret}\ V_i'[\gamma_p]$ for some V_1', V_2' such that $V_i \rightsquigarrow^{CT} \texttt{ret}\ V_i'$. Then by Lemma 208, and reducing we get

$$\begin{array}{ll} & M_1'[\gamma_p] \\ \mapsto^+ & (a \leftarrow [\![A_1^{\sqsubseteq}]\!]\updownarrow[\texttt{ret}\ V_2'];[\![A_2^{\sqsubseteq}]\!]\updownarrow[\texttt{force}\ V_1'\ a])[\gamma_p] \end{array}$$

  So the result follows by context plugging and the definition of the translation relation.

- tag downcast success:

$$E[\langle \text{tag}_G(A^{\sqsubseteq})\rangle\downarrow\, \langle \text{tag}_G(G)\rangle\updownarrow\, V] \mapsto E[\langle A^{\sqsubseteq}\rangle\downarrow\, V]$$

  We have $M_1'[\gamma_p] \cong_b S'[[\![\text{tag}_G(A^{\sqsubseteq})]\!]\downarrow[r \leftarrow N';\texttt{ret}\ \texttt{inj}_{\text{case}(G)}\ r][\gamma_p]]$ where $V \rightsquigarrow^{CT} N', E \rightsquigarrow_s^{CT} S'$. Then, by Lemma 204 (value translation), we have some V' such that $N'[\gamma_p] \mapsto_b^* \texttt{ret}\ V'[\gamma_p]$ and $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. Thus, reducing and using Lemma 208,

$$\begin{array}{ll} & [\![M]\!]\downarrow_1'[\gamma_p] \\ \mapsto^+ & S[[\![A^{\sqsubseteq}]\!]\downarrow[\texttt{ret}\ V']][\gamma_p] \end{array}$$

  and the result follows by substitution, plugging lemmas and definition of the translation relation.

- tag downcast error:

$$E[\langle \text{tag}_G(A^{\sqsubseteq})\rangle\downarrow\, \langle \text{tag}_H(H)\rangle\updownarrow\, V] \mapsto E[\mathcal{U}_B] \text{ where } G \neq H$$

  We have $M_1' \cong_b S'[[\![\text{tag}_G(A^{\sqsubseteq})]\!]\downarrow[r \leftarrow N';\texttt{ret}\ \texttt{inj}_{\text{case}(H)}\ r][\gamma_p]]$ where $V \rightsquigarrow^{CT} N'$, $E \rightsquigarrow_s^{CT} S'$. Then, by Lemma 204 (value translation), we have some V' such that $N'[\gamma_p] \mapsto_b^* \texttt{ret}\ V'[\gamma_p]$ and $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. Thus, by Lemma 208 and reducing:

$$\begin{array}{ll} & M_1'[\gamma_p] \\ \mapsto^+ & S[\mathcal{U}][\gamma_p] \end{array}$$

  And the result follows by plugging and definition of the translation relation.

- Existential upcast:

$$E[\langle \exists^{\nu} X.A_1^{\sqsubseteq}\rangle\updownarrow\, \texttt{pack}^{\nu}(X \cong A, [A_2^{\sqsubseteq}\updownarrow ...], N_1)] \mapsto E[\texttt{pack}^{\nu}(X \cong A, [A_1^{\sqsubseteq}\updownarrow, A_2^{\sqsubseteq}\updownarrow ...], N_1)]$$

For some N_1' such that $N_1 \rightsquigarrow^{CT} N_1'$, we have

$$M_1'[\gamma_p] \cong_b S'[[\![\exists^\nu X.A_1^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{ret pack}(A', \texttt{thunk } \lambda c_X : \text{Case } A'.N'_{cst}) \texttt{ as } [\![\Sigma; \cdot \vdash \exists^\nu X.A_{1r}^{\sqsubseteq}]\!]]][\gamma_p]$$

where $N_{cst} = [\![A_2^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force } (\texttt{thunk } (\lambda c_X : \text{Case } A'.(\ldots N_1' \ldots))) \ c_X]$ and $A' = [\![\Sigma; \cdot \vdash A]\!]$ and $E \rightsquigarrow_s^{CT} S'$. By Lemma 208, reducing we get:

$$S'[M_1'][\gamma_p] \mapsto^+ \texttt{ret pack}(A', \texttt{thunk } \lambda c_X : \text{Case } A'. [\![A_1^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force } (\texttt{thunk } \lambda c_X : \text{Case } A'.N'_{cst}) \ c_X]) \texttt{ as } [\![\exists^\nu X.A_{1r}^{\sqsubseteq}]\!][\gamma_p]$$

And the result follows by substitution/plugging lemmas and definition of the translation.

- Universal cast:

$$E[\langle \forall^\nu X.A^{\sqsubseteq} \rangle^{\updownarrow} (\Lambda^\nu\{X.([\overline{B^{\sqsubseteq} \updownarrow}], M)\})] \mapsto E[\Lambda^\nu\{X.([A^{\sqsubseteq} \updownarrow, \overline{B^{\sqsubseteq} \updownarrow}], M)\}]$$

First, let $V = \Lambda^\nu\{X.([\overline{B^{\sqsubseteq} \updownarrow}], M)\}$. By Lemma 208, and reducing, we get

$$M_1'[\gamma_p] \cong_b S'[x \leftarrow N'; \texttt{ret thunk } (\Lambda X.\lambda c_X : \text{Case } X.[\![A^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force } x \ [X] \ c_X])]$$

where $V \rightsquigarrow^{CT} N'$, $E \rightsquigarrow_s^{CT} S'$. By the value reduction lemma, $N' \mapsto_b^* \texttt{ret } V'$ for some $V \rightsquigarrow^{CT} \texttt{ret } V'$.

Reducing, we get

$$S'[x \leftarrow N'; \texttt{ret thunk } (\Lambda X.\lambda c_X : \text{Case } X.[\![A^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force } x \ [X] \ c_X])] \mapsto_b^* S'[\texttt{ret thunk } (\Lambda X.\lambda c_X : \text{Case } X.[\![A^{\sqsubseteq}]\!]^{\updownarrow}[\texttt{force } V' \ [X] \ c_X])]$$

Call this term N_2' and note that $E[\Lambda^\nu\{X.([A^{\sqsubseteq} \updownarrow, \overline{B^{\sqsubseteq} \updownarrow}], M)\}] \rightsquigarrow^{CT} N_2'$. Then by bind confluence, $M_1', N_2' \mapsto_b^* M_2'$ and by closure under bind reduction, we need only show that the size of the source term is reduced. This holds because

$$|E[\langle \forall^\nu X.A^{\sqsubseteq} \rangle^{\updownarrow} (\Lambda^\nu\{X.([\overline{B^{\sqsubseteq} \updownarrow}], M)\})]| = 2 + |E[(\Lambda^\nu\{X.([\forall^\nu X.A^{\sqsubseteq} \updownarrow, \overline{B^{\sqsubseteq} \updownarrow}], M)\})]|$$

- Tag check true:

$$E[\texttt{is}(G)? \ \langle \text{tag}_G(G) \rangle^{\updownarrow} \ V] \mapsto E[\texttt{true}]$$

We have

$$M_1'[\gamma_p] \cong_b S'[(r \leftarrow (x \leftarrow N'; \texttt{ret inj}_{\text{case}(G)} \ x); \texttt{match case}(G) \texttt{ with } r\{\texttt{inj } y.\texttt{ret true} \mid \texttt{ret false}\})][\gamma_p]$$

where $V \rightsquigarrow^{CT} N'$. Then, by Lemma 204 (value translation), we have some V' such that $N'[\gamma_p] \mapsto_b^* \texttt{ret}\ V'[\gamma_p]$ and $V \rightsquigarrow^{CT} \texttt{ret}\ V'$. So by Lemma 208, have the following reduction:

$$M_1'[\gamma_p] \mapsto^+ \texttt{ret}\ S'[\texttt{true}]$$

Since $\texttt{true} \rightsquigarrow^{CT} \texttt{ret true}$, we conclude by Lemma 198 (translation context plug).

- Tag check false:

$$E[\texttt{is}(G)?\ \langle \text{tag}_H(H)\rangle\!\uparrow V] \mapsto E[\texttt{false}] \text{ where } G \neq H$$

This case is analogous to the former except that since $G \neq H$, the translation produces $\texttt{false}$ and since $\texttt{false} \rightsquigarrow^{CT} \texttt{false}$, we conclude.

□

Now that we have established the simulation theorem, we can prove our desired adequacy theorems that say that we can tell if a PolyC$^\nu$ term terminates, errors or diverges by looking at its translation to CBPV. They follow by our simulation theorem and progress for PolyC$^\nu$.

Corollary 213. *If* $\Sigma; \cdot \vdash M : A$ *and* $M \rightsquigarrow^{CT} M'$, *then*

- *If* $M \mapsto^* V$, *then* $M'[\gamma_p] \mapsto^* \texttt{ret}\ V'[\gamma_p]$ *with* $V \rightsquigarrow^{CT} \texttt{ret}\ V'$.
- *If* $M \mapsto^* \mathcal{U}$, *then* $M'[\gamma_p] \mapsto^* \mathcal{U}$

Proof.

- By simulation, there is some N' with $V \rightsquigarrow^{CT} N'$ and then by Lemma 204.
- By simulation there is some N' with $M'[\gamma_p] \mapsto^* N'[\gamma_p]$ and $\mathcal{U} \rightsquigarrow^{CT} N'$. By definition, $N' = \mathcal{U}$.

□

Corollary 214. *If* $\Sigma; \cdot \vdash M : A$ *and* $M \rightsquigarrow^{CT} M'$, *and* $\Sigma_p, [\![\Sigma]\!] \triangleright M'[\gamma_p] \Uparrow$, *then* $\Sigma \triangleright M \Uparrow$.

Proof. Assume to the contrary that M terminates, then by Lemma 190, M evaluates to a value or error, in which case the previous corollary proves $M'[\gamma_p]$ terminates, which is a contradiction. □

Corollary 215. *If* $\Sigma; \cdot \vdash M : A$ *and* $M \rightsquigarrow^{CT} M'$, *and* $\Sigma \triangleright M \Uparrow$, *then* $\Sigma_p, [\![\Sigma]\!] \triangleright M'[\gamma_p] \Uparrow$.

Proof. Coinductively it is sufficient to show that if $M \rightsquigarrow^{CT} M'$ and $M \Uparrow$ then $M \mapsto N$, $M'[\gamma_p] \mapsto^+ N'[\gamma_p]$ and $N \rightsquigarrow^{CT} N'$.

We proceed by induction on $|M|$. Since $M \Uparrow$, $M \mapsto N$ and $N \Uparrow$. By the simulation lemma, either $M'[\gamma_p] \mapsto^* N'[\gamma_p]$ where $N \rightsquigarrow^{CT} N'$ and one of two cases holds:

- If $M'[\gamma_p] \mapsto^+ N'[\gamma_p]$ we're done.
- If $|N| < |M|$ the result follows by inductive hypothesis.

□

Theorem 216. *If* $\Sigma; \cdot \vdash M : A$ *and* $M \rightsquigarrow^{CT} M'$,

- *If* $M'[\gamma_p] \mapsto^* \texttt{ret}\ V'$, *then* $M \mapsto^* V$ *for some* V.
- *If* $M'[\gamma_p] \mapsto^* \mho$, *then* $M \mapsto^* \mho$.

Proof. Assume to the contrary that M' terminates but M diverges. But then by the previous corollary M' diverges, which is a contradiction. Therefore M either errors or runs to a value, and it follows by determinacy of reduction and the forward reasoning that M must match M's behavior. □

In §10.6, we establish graduality and parametricity theorems for PolyG^v/PolyC^v by analysis of the semantics of terms translated into CBPV$_{\text{OSum}}$. But since we take the operational semantics of PolyC^v as definitional, we need to bridge the gap between the operational semantics in CBPV$_{\text{OSum}}$ and PolyC^v by proving the following adequacy theorem that says that the final behavior of terms in PolyC^v is the same as the behavior of their translations:

Theorem 217 (Adequacy)**.** *If* $\cdot \vdash M : A; \cdot$, *then*

- $M \Uparrow$ *if and only if* $[\![M]\!][\gamma_p] \Uparrow$.
- $M \mapsto^* \mho$ *if and only if* $[\![M]\!][\gamma_p] \mapsto^* \mho$.
- $M \mapsto^* \texttt{ret}\ V$ *if and only if* $[\![M]\!][\gamma_p] \mapsto^* \texttt{ret}\ V'$ *for some* V'.

Proof. By the previous corollaries, since $M \rightsquigarrow^{CT} [\![M]\!]$. □

10.6 GRADUALITY AND PARAMETRICITY

In this section we prove the central metatheoretic results of the paper: that our surface language satisfies both graduality and parametricity theorems. Each of these is considered a technical challenge to prove: parametricity is typically proven by a logical relation and graduality is proven either by a simulation argument [75] or a logical relation [56, 60], so in the worst case this would require two highly technical developments. However, we show that this is not necessary: the logical relations proof for graduality is general enough that the parametricity theorem is a corollary of the *reflexivity* of the logical relation. This substantiates the analogy between parametricity and graduality originally proposed in [56].

The key to sharing this work is that we give a novel *relational* interpretation of type precision derivations. That is, our logical relation

is indexed not by types, but by type precision derivations. For any derivation $A^{\sqsubseteq} : A_l \sqsubseteq A_r$, we define a relation $\mathcal{V}[\![A^{\sqsubseteq}]\!]$ between values of A_l and A_r. By taking the reflexivity case $A : A \sqsubseteq A$, we recover the parametricity logical relation. Previous logical relations proofs of graduality defined a logical relation indexed by types, and used casts to define a second relation based on type precision judgments, but the direct relational approach simplifies the proofs and immediately applies to parametricity as well.

10.6.1 *Term Precision*

To state the graduality theorem, we begin by formalizing the syntactic *term* precision relation. The intuition behind a precision relation $M \sqsubseteq M'$ is that M' is a (somewhat) dynamically typed term and we have changed some of its type annotations to be more precise, producing M. This is one of the main intended use cases for a gradual language: hardening the types of programs over time. Restated in a less directed way, a term M is (syntactically) more precise than M' when the types and annotations in M are more precise than M' and otherwise the terms have the same structure. We formalize this as a judgment $\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}$, with the following somewhat intricate side conditions:

- $\Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r$ is a well-formed type precision*context*,
- $\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$ is a well-formed type precision derivation,
- M_l and M_r is are well-typed with typings $\Gamma_l \vdash M_l : A_l$ and $\Gamma_r \vdash M_r : A_r$

A precision context $\Gamma^{\sqsubseteq}$ is like a precision derivation between two contexts: everywhere a type would be in an ordinary context, a precision derivation is used instead. Well formedness of precision contexts, and the extension of well-formedness of type precision derivations to type precision contexts are straightforward, and presented in Figure 10.22.

We show term precision rules for the surface language in Figure 10.24. The rules are all completely structural: just check that the two terms have the same term constructor and all of the corresponding arguments of the rule are $\sqsubseteq$. As exhibited by the $\forall^{\nu}$ elimination rule, the metafunctions $\mathrm{dom}, \mathrm{cod}, \mathrm{un}\forall^{\nu}, \mathrm{un}\exists^{\nu}$ are extended in the obvious way to work on precision derivations.

Figure 10.24 shows the full definition of term precision for PolyC$^{\nu}$. The main difference is that, following [60], we include four rules involving casts: two for downcasts and two for upcasts. We can summarize all four by saying that if $M_l \sqsubseteq M_r$, then adding a cast to either M_l or M_r still maintains that the left side is more precise than the right, as long as the type on the left is more precise than the right. Semantically, these are the most important term precision rules, as they bridge the worlds of type and term precision.

$$\cdot : \cdot \sqsubseteq \cdot \qquad \frac{\Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{(\Gamma^{\sqsubseteq}, x : A^{\sqsubseteq}) : \Gamma_l, x : A_l \sqsubseteq \Gamma_r, x : A_r}$$

$$\frac{\Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{(\Gamma^{\sqsubseteq}, X \cong A^{\sqsubseteq}) : \Gamma_l, X \cong A_l \sqsubseteq \Gamma_r, X \cong A_r} \qquad \frac{\Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r}{(\Gamma^{\sqsubseteq}, X) : \Gamma_l, X \sqsubseteq \Gamma_r, X}$$

$$\Gamma^{\sqsubseteq} \vdash ? : ? \sqsubseteq ? \qquad \Gamma^{\sqsubseteq} \vdash \texttt{Bool} : \texttt{Bool} \sqsubseteq \texttt{Bool} \qquad \frac{X \in \Gamma^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash X : X \sqsubseteq X}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A \sqsubseteq G}{\Gamma^{\sqsubseteq} \vdash \mathrm{tag}_G(A^{\sqsubseteq}) : A \sqsubseteq ?} \qquad \frac{\Gamma^{\sqsubseteq}, X \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma^{\sqsubseteq} \vdash \forall^\nu X.A^{\sqsubseteq} : \forall^\nu X.A_l \sqsubseteq \forall^\nu X.A_r}$$

$$\frac{\Gamma^{\sqsubseteq}, X \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma^{\sqsubseteq} \vdash \exists^\nu X.A^{\sqsubseteq} : \exists^\nu X.A_l \sqsubseteq \exists^\nu X.A_r}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} \to B^{\sqsubseteq} : A_l \to B_l \sqsubseteq A_r \to B_r}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} \times B^{\sqsubseteq} : A_l \times B_l \sqsubseteq A_r \times B_r}$$

Figure 10.22: Type Precision Contexts, Type Precision Derivations in Context

$$\begin{aligned}
\mathrm{cod}(A^{\sqsubseteq} \to B^{\sqsubseteq}) &= B^{\sqsubseteq} \\
\mathrm{cod}(?) &= ? \\
\mathrm{cod}(\mathrm{tag}_G(A^{\sqsubseteq} \to B^{\sqsubseteq})) &= B^{\sqsubseteq} \\
\pi_i(A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}) &= A_i^{\sqsubseteq} \\
\pi_i(?) &= ? \\
\pi_i(\mathrm{tag}_{?\times?}(A_0^{\sqsubseteq} \times A_1^{\sqsubseteq})) &= A_i^{\sqsubseteq} \\
\\
\mathrm{un}\forall^\nu(\forall^\nu X.A^{\sqsubseteq}) &= A^{\sqsubseteq} \\
\mathrm{un}\forall^\nu(?) &= ? \\
\mathrm{un}\forall^\nu(\mathrm{tag}_{\forall^\nu X.?}(\forall^\nu X.A^{\sqsubseteq})) &= A^{\sqsubseteq} \\
\mathrm{un}\exists^\nu(\exists^\nu X.A^{\sqsubseteq}) &= A^{\sqsubseteq} \\
\mathrm{un}\exists^\nu(?) &= ? \\
\mathrm{un}\exists^\nu(\mathrm{tag}_{\exists^\nu X.?}(\exists^\nu X.A^{\sqsubseteq})) &= A^{\sqsubseteq}
\end{aligned}$$

Figure 10.23: Metafunctions extended to type precision derivations

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma^{\sqsubseteq} \vdash (M_l :: B_l) \sqsubseteq (M_r :: B_r) : B^{\sqsubseteq}} \qquad \frac{x : A^{\sqsubseteq} \in \Gamma^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash x \sqsubseteq x : A^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, x : A^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ x = M_l; N_l \sqsubseteq \texttt{let}\ x = M_r; N_r : B}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad X \cong B^{\sqsubseteq} \in \Gamma^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{seal}_X M_l \sqsubseteq \texttt{seal}_X M_r : X}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad X \cong B^{\sqsubseteq} \in \Gamma^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{unseal}_X M_l \sqsubseteq \texttt{unseal}_X M_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash G}{\Gamma^{\sqsubseteq} \vdash \texttt{is}(G)\texttt{?}\ M_l \sqsubseteq \texttt{is}(G)\texttt{?}\ M_r : \texttt{Bool}} \qquad \Gamma^{\sqsubseteq} \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool}$$

$$\Gamma^{\sqsubseteq} \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool}$$

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \\ \Gamma^{\sqsubseteq} \vdash N_{lt} \sqsubseteq N_{rt} : B_t^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash N_{lf} \sqsubseteq N_{rf} : B_f^{\sqsubseteq}\end{array}}{\Gamma^{\sqsubseteq} \vdash \texttt{if}\ M_l\ \texttt{then}\ N_{lt}\ \texttt{else}\ N_{lf} \sqsubseteq \texttt{if}\ M_r\ \texttt{then}\ N_{rt}\ \texttt{else}\ N_{rf} : B_t^{\sqsubseteq} \sqcup B_f^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_{l1} \sqsubseteq M_{r1} : A_1^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash M_{l2} \sqsubseteq M_{r2} : A_2^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash (M_{l1}, M_{l2}) \sqsubseteq (M_{r1}, M_{r2}) : A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, x : \pi_0(A^{\sqsubseteq}), y : \pi_1(A_1^{\sqsubseteq}) \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ (x, y)\ =\ M_l; N_l \sqsubseteq \texttt{let}\ (x, y)\ =\ M_r; N_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq}, x : A^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma^{\sqsubseteq} \vdash \lambda x : A_l.M_l \sqsubseteq \lambda x : A_r.M_r : A^{\sqsubseteq} \to B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash M_l\ N_l \sqsubseteq M_r\ N_r : \mathrm{cod}(A^{\sqsubseteq})}$$

$$\frac{\Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma^{\sqsubseteq} \vdash \texttt{pack}^{\nu}(X \cong B_l, M_l) \sqsubseteq \texttt{pack}^{\nu}(X \cong B_r, M_r) : \exists^{\nu} X.A^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, X, x : \mathrm{un}\exists^{\nu}(A^{\sqsubseteq}) \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{unpack}\ (X, x) = M_l; N_l \sqsubseteq \texttt{unpack}\ (X, x) = M_r; N_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq}, X \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \Lambda^{\nu} X.M_l \sqsubseteq \Lambda^{\nu} X.M_r : \forall^{\nu} X.A^{\sqsubseteq}}$$

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad B^{\sqsubseteq} : B_l \sqsubseteq B_r \\ \Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq}, x : \mathrm{un}\forall^{\nu}{}_X(A^{\sqsubseteq}) \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq\prime}\end{array}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ x = M_l\{X \cong B_l\}; N_l \sqsubseteq \texttt{let}\ x = M_r\{X \cong B_r\}; N_r : B^{\sqsubseteq\prime}}$$

Figure 10.24: PolyG$^{\nu}$ Term Precision

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : AC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \quad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma_l \vdash AB^{\sqsubseteq} : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vdash BC^{\sqsubseteq} : B \sqsubseteq C}{\Gamma^{\sqsubseteq} \vdash \langle AB^{\sqsubseteq}\rangle\!\uparrow M_l \sqsubseteq M_r : BC^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : BC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \quad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma_l \vdash AB^{\sqsubseteq} : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vdash BC^{\sqsubseteq} : B \sqsubseteq C}{\Gamma^{\sqsubseteq} \vdash \langle AB^{\sqsubseteq}\rangle\!\downarrow M_l \sqsubseteq M_r : AC^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : AB^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \quad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma_r \vdash BC^{\sqsubseteq} : B \sqsubseteq C \qquad \Gamma^{\sqsubseteq}, \Gamma^{\sqsubseteq\prime} \vdash AB^{\sqsubseteq} : A \sqsubseteq B}{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq \langle BC^{\sqsubseteq}\rangle\!\uparrow M_r : AC^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : AC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \quad \Gamma^{\sqsubseteq}, \Gamma^{\sqsubseteq\prime} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma_r \vdash BC^{\sqsubseteq} : B \sqsubseteq C \qquad \Gamma^{\sqsubseteq}, \Gamma^{\sqsubseteq\prime} \vdash AB^{\sqsubseteq} : A \sqsubseteq B}{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq \langle BC^{\sqsubseteq}\rangle\!\downarrow M_r : AB^{\sqsubseteq}}$$

Figure 10.25: PolyC$^{\nu}$ Term Precision Part 1

Then the key lemma is that the elaboration process from PolyG$^{\nu}$ to PolyC$^{\nu}$ preserves term precision:

Lemma 218. *If* $\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}$ *in the surface language, then* $\Gamma^{\sqsubseteq} \vdash M_l^+ \sqsubseteq M_r^+ : A^{\sqsubseteq}$

To establish this we need to prove a few supporting lemmas. First, we need some basic facts about type precision derivations.

First, there is at most one derivation of any judgment $A \sqsubseteq A'$

Lemma 219. *If* $\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A \sqsubseteq A'$ *and* $\Gamma^{\sqsubseteq} B^{\sqsubseteq} : A \sqsubseteq A'$ *then* $A^{\sqsubseteq} = B^{\sqsubseteq}$

Proof. By induction on derivations. If $A' \neq ?$, then only one rule applies and the proof follows by inductive hypotheses. If $A^{\sqsubseteq} = ? : ? \sqsubseteq ?$, the only other rule that could apply is $\mathrm{tag}_G(A^{\sqsubseteq})$, which cannot apply because it is easy to see that $? \sqsubseteq G$ does not hold for any G. Finally, we need to show that if $A^{\sqsubseteq} = \mathrm{tag}_G(A^{\sqsubseteq})$ and $B^{\sqsubseteq} = \mathrm{tag}_{G'}(A^{\sqsubseteq})$ then $G = G'$ because it is easy to see if $A \sqsubseteq G$ and $A \sqsubseteq G'$ then $G = G'$. □

Next, reflexivity.

Lemma 220. *If* $\Gamma \vdash A$ *is a well-formed type then* $\Gamma \vdash A : A \sqsubseteq A$.

Proof. By induction over A. Every type constructor is punned with its type precision constructor. □

Then, transitivity.

Lemma 221. *If* $\Gamma^{\sqsubseteq} \vdash AB^{\sqsubseteq} : A \sqsubseteq B$ *and* $\Gamma^{\sqsubseteq} \vdash BC^{\sqsubseteq} : B \sqsubseteq C$ *then we can construct a proof* $\Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C$.

Proof. By induction on $BC^{\sqsubseteq}$.

$$\frac{x : A^{\sqsubseteq} \in \Gamma^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash x \sqsubseteq x : A^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, x : A^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ x = M_l; N_l \sqsubseteq \texttt{let}\ x = M_r; N_r : B^{\sqsubseteq}}$$

$$\frac{(X \cong A^{\sqsubseteq}) \in \Gamma^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{seal}_X M_l \sqsubseteq \texttt{seal}_X M_r : X}$$

$$\frac{(X \cong A^{\sqsubseteq}) \in \Gamma^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : X}{\Gamma^{\sqsubseteq} \vdash \texttt{unseal}_X M_l \sqsubseteq \texttt{unseal}_X M_r : A^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : ? \qquad \Gamma^{\sqsubseteq} \vdash G}{\Gamma^{\sqsubseteq} \vdash \texttt{is}(G)?\ M_l \sqsubseteq \texttt{is}(G)?\ M_r : \texttt{Bool}} \qquad \Gamma^{\sqsubseteq} \vdash \texttt{true} \sqsubseteq \texttt{true} : \texttt{Bool}$$

$$\Gamma^{\sqsubseteq} \vdash \texttt{false} \sqsubseteq \texttt{false} : \texttt{Bool}$$

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : \texttt{Bool} \\ \Gamma^{\sqsubseteq} \vdash N_{lt} \sqsubseteq N_{rt} : B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash N_{lf} \sqsubseteq N_{rf} : B^{\sqsubseteq}\end{array}}{\Gamma^{\sqsubseteq} \vdash \texttt{if}\ M_l\ \texttt{then}\ N_{lt}\ \texttt{else}\ N_{lf} \sqsubseteq \texttt{if}\ M_r\ \texttt{then}\ N_{rt}\ \texttt{else}\ N_{rf} : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_{l1} \sqsubseteq M_{r1} : A_1^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash M_{l2} \sqsubseteq M_{r2} : A_2^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash (M_{l1}, M_{l2}) \sqsubseteq (M_{r1}, M_{r2}) : A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A_1^{\sqsubseteq} \times A_2^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, x : A_1^{\sqsubseteq}, y : A_2^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ (x, y)\ =\ M_l; N_l \sqsubseteq \texttt{let}\ (x, y)\ =\ M_r; N_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq}, x : A^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma^{\sqsubseteq} \vdash \lambda x : A_l.M_l \sqsubseteq \lambda x : A_l.M_l : A^{\sqsubseteq} \to B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq} \to B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash M_l\, N_l \sqsubseteq M_r\, N_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \texttt{pack}^{\nu}(X \cong B_l, M_l) \sqsubseteq \texttt{pack}^{\nu}(X \cong B_r, M_r) : \exists^{\nu} X.A^{\sqsubseteq}}$$

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : \exists^{\nu} X.A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} \\ \Gamma^{\sqsubseteq}, X, x : A^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}\end{array}}{\Gamma^{\sqsubseteq} \vdash \texttt{unpack}\ (X, x) = M_l; N_l \sqsubseteq \texttt{unpack}\ (X, x) = M_r; N_r : B^{\sqsubseteq}}$$

$$\frac{\Gamma^{\sqsubseteq}, X \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \Lambda^{\nu} X.M_l \sqsubseteq \Lambda^{\nu} X.M_r : \forall^{\nu} X.A^{\sqsubseteq}}$$

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : \forall^{\nu} X.A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r \\ \Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq}, x : A^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq\prime}\end{array}}{\Gamma^{\sqsubseteq} \vdash \texttt{let}\ x = M_l\{X \cong B_l\}; N_l \sqsubseteq \texttt{let}\ x = M_l\{X \cong B_l\}; N_l : B^{\sqsubseteq\prime}}$$

Figure 10.26: PolyC$^{\nu}$ Term Precision Part 2

1. If $BC^{\sqsubseteq} = ? : ? \sqsubseteq ?$, then the proof is just $AB^{\sqsubseteq}$
2. If $BC^{\sqsubseteq} = \mathrm{tag}_G(BG^{\sqsubseteq})$, then $BG^{\sqsubseteq} : B \sqsubseteq G$ so by inductive hypothesis, there is a proof $AG^{\sqsubseteq} : A \sqsubseteq G$ and the proof we need is $\mathrm{tag}_G(AG^{\sqsubseteq})$.
3. If $BC^{\sqsubseteq} = BC_1^{\sqsubseteq} \times BC_2^{\sqsubseteq}$, then it must also be the case that $AB^{\sqsubseteq} = AB_1^{\sqsubseteq} \times AB_2^{\sqsubseteq}$, and then our result is $AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}$ where $AC_1^{\sqsubseteq}, AC_2^{\sqsubseteq}$ come from the inductive hypothesis.
4. All other cases are analogous to the product.

□

And we prove the gradual meet, when it exists is in fact the greatest lower bound in the precision ordering $\sqsubseteq$.

Lemma 222. *For every $\Gamma \vdash A, B$, $\Gamma \vdash A \sqcap B$ and there are precision derivations*

1. $\Gamma \vdash A\sqcap^{\sqsubseteq} : A \sqcap B \sqsubseteq A$
2. $\Gamma \vdash B\sqcap^{\sqsubseteq} : A \sqcap B \sqsubseteq B$

Such that for any $\Gamma \vdash C$ with $\Gamma \vdash CA^{\sqsubseteq} : C \sqsubseteq A$ and $\Gamma \vdash CB^{\sqsubseteq} : C \sqsubseteq B$, there exists a derivation

$$\Gamma \vdash C\sqcap^{\sqsubseteq} : C \sqsubseteq A \sqcap B$$

Then the only complex case of the proof is to show that the casts are monotone, since all other cases will be primitive rules.

Lemma 223 (Casts are Monotone)**.** *If $A^{\sqsubseteq} : A_l \sqsubseteq A_r$ and $AB_l^{\sqsubseteq} : A_l \sqsubseteq B_l$ and $AB_r^{\sqsubseteq} : A_r \sqsubseteq B_r$ and $B^{\sqsubseteq} : B_l \sqsubseteq B_r$, then*

1. *If $\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}; \Gamma^{\sqsubseteq}$, then $\Gamma^{\sqsubseteq} \vdash \langle AB_l^{\sqsubseteq}\rangle\!\updownarrow M_l \sqsubseteq \langle AB_r^{\sqsubseteq}\rangle\!\updownarrow M_r : B^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}$*
2. *If $\Gamma^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}$, then $\Gamma^{\sqsubseteq} \vdash \langle c\rangle\!\downarrow$ $insertsthat AB_l^{\sqsubseteq} N_l \sqsubseteq \langle AB_r^{\sqsubseteq}\rangle\!\downarrow N_r : A^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}$*

Proof. 1. By the following derivation

$$\dfrac{\dfrac{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq \langle AB_r^{\sqsubseteq}\rangle\!\updownarrow M_r : AB_{lr}^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}}{\Gamma^{\sqsubseteq} \vdash \langle AB_l^{\sqsubseteq}\rangle\!\updownarrow M_l \sqsubseteq \langle AB_r^{\sqsubseteq}\rangle\!\updownarrow M_r : B^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}$$

Where $AB_{lr}^{\sqsubseteq} : A_l \sqsubseteq B_r$, which exists by transitivity Lemma 221.

2. By the following derivation

$$\dfrac{\dfrac{\Gamma^{\sqsubseteq} \vdash N_l \sqsubseteq N_r : B^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vdash \langle AB_l^{\sqsubseteq}\rangle\!\downarrow N_l \sqsubseteq N_r : AB_{lr}^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}}{\Gamma^{\sqsubseteq} \vdash \langle AB_l^{\sqsubseteq}\rangle\!\downarrow N_l \sqsubseteq \langle AB_r^{\sqsubseteq}\rangle\!\downarrow N_r : A^{\sqsubseteq}; \Gamma_o^{\sqsubseteq}}$$

Where $AB_{lr}^{\sqsubseteq} : A_l \sqsubseteq B_r$, which exists by transitivity Lemma 221.

□

And finally we can prove the monotonicity of the elaboration.

PROOF OF LEMMA 218

Proof. By induction on term precision derivations.

1. Cast. By applying Lemma 223 twice.
2. Var: Immediate
3. Let: immediate
4. seal: by the argument for the ascription case.
5. unseal: There are three cases for $A^{\sqsubseteq}$: X, ? and $\mathrm{tag}_X(X)$. The first case is immediate. If $A^{\sqsubseteq} = ?$, we need to show

   $$\mathtt{unseal}_X\langle \mathrm{tag}_X(X)\rangle\downarrow M_l^+ \sqsubseteq \mathtt{unseal}_X\langle \mathrm{tag}_X(X)\rangle\downarrow M_r^+$$

   Which follows by $\mathtt{unseal}_X$ congruence and Lemma 223. For the final case we need to show

   $$\mathtt{unseal}_X M_l^+ \sqsubseteq \mathtt{unseal}_X\langle \mathrm{tag}_X(X)\rangle\downarrow M_r^+$$

   which follows by congruence for $\mathtt{unseal}_X$ and the downcast-right rule.
6. tag-check $\mathtt{is}(G)?\ M_l \sqsubseteq \mathtt{is}(G)?\ M_r$: follows by congruence and Lemma 223.
7. tru: Immediate
8. fls: Immediate
9. if: By if congruence, we need to show the condition and the two branches of the if are ordered.
   - For the condition there are three subcases $M_l \sqsubseteq M_r : A^{\sqsubseteq}$: either $\mathtt{Bool}$, ? or $\mathrm{tag}_{\mathtt{Bool}}(\mathtt{Bool})$. The ordering follows by the same argument as the $\mathtt{unseal}_X$ case.
   - The two branches follow by the same argument. We describe the true branch. We have by inductive hypothesis that $N_{tl}^+ \sqsubseteq N_{tr}^+$ and we need to show

   $$\langle B_{tl}^{\sqsubseteq}\rangle\downarrow \mathtt{hide}\ \Gamma_{tl} \subseteq \Gamma_{tl} \cap \Gamma_{fl}; N_{tl}^+ \sqsubseteq \langle B_{tr}^{\sqsubseteq}\rangle\downarrow \mathtt{hide}\ \Gamma_{tr} \subseteq \Gamma_{tr} \cap \Gamma_{fr}; N_{tr}^+$$

   Which follows by lemmas 223.
10. Pair intro: Immediate by inductive hypothesis.
11. Pair elim: By similar argument to $\mathtt{unseal}_X$
12. Function application: similar argument to $\mathtt{unseal}_X$

13. $\exists^{\nu}$ introduction: by congruence
14. $\forall^{\nu}$ elimination: by similar argument to $\texttt{unseal}_X$.
15. $\forall^{\nu}$ introduciton: by congruence.
16. $\forall^{\nu}$ elimination: by similar argument to $\texttt{unseal}_X$ case.

□

10.6.2 *Graduality Theorem*

The graduality theorem states that if a term M is *syntactically* more precise than a term M', then M *semantically* refines the behavior of M': it may error, but otherwise it has the same behavior as M': if it diverges so does M' and if it terminates at V, M' terminates with some V' as well. If we think of M as the result of hardening the types of M', then this shows that hardening types semantically only increases the burden of runtime type checking and doesn't otherwise interfere with program behavior. We call this *operational* graduality, as we will consider some related notions later.

Theorem 224 (Operational Graduality)**.** *If* $\cdot \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}$*, then either* $M_l^+ \Downarrow \upsilon$ *or both terms diverge* $M_l^+, M_r^+ \Uparrow$ *or both terms terminate successfully* $M_l^+ \Downarrow V_l$ *and* $M_r^+ \Downarrow V_r$*.*

10.6.3 *Logical Relation*

The basic idea of the logical relations proof to proving graduality is to interpret a term precision judgment $\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}$ in a *relational* manner. That is, to every type precision derivation $A^{\sqsubseteq} : A_l \sqsubseteq A_r$, we associate a relation $\mathcal{V}[\![A^{\sqsubseteq}]\!]$ between closed values of types A_l and A_r. Then we define a semantic version of the term precision judgment $\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq}$ which says that given inputs satisfying the relations in $\Gamma^{\sqsubseteq}$, then either M_l will error, both sides diverge, or M_l and M_r will terminate with values in the relation $\mathcal{V}[\![A^{\sqsubseteq}]\!]$. We define this relation over $\text{CBPV}_{\text{OSum}}$ translations of PolyC^{ν} terms, rather than PolyC^{ν} terms because the operational semantics is simpler.

More precisely, we use the now well established technique of Kripke, step-indexed logical relations [3]. Because the language includes allocation of fresh type names at runtime, the set of values that should be in the relation grows as the store increases. This is modeled *Kripke* structure, which indexes the relation by a "possible world" that attaches invariants to the allocated cases. Because our language includes diverging programs (due to the open sum type), we need to use a *step-indexed* relation that decrements when pattern matching on OSum, and "times out" when the step index hits 0. Finally, following [56, 60], to model graduality we need to associate two relations to each type

precision derivation: one which times out when the left hand hand term runs out of steps, but allows the right hand side to take any number of steps and vice-versa one that times out when the right runs out of steps.

Figure 10.27 includes preliminary definitions we need for the logical relation. First, $\mathrm{Atom}_n[A_l, A_r]$ and $\mathrm{CAtom}_n[A_l, A_r]$ define the world-term-term triples that the relations are defined over. A relation $R \in \mathrm{Rel}_n[A_l, A_r]$ at stage n consists of triples of a world, and a value of type A_l and a value of type A_r (ignore the n for now) such that it is monotone in the world. The world $w \in \mathrm{World}_n$ contains the number of steps remaining $w.j$, the current state of each side $w.\Sigma_l$, $w.\Sigma_r$, and finally an interpretation of how the cases in the two stores are related $w.\eta$. An interpretation $\eta \in \mathrm{Interp}_n[\Sigma_l, \Sigma_r]$ consists of a cardinality $\eta.size$ which says how many cases are related and a function $\eta.f$ which says *which* cases are related, i.e., for each $i \in \eta.size$ it gives a pair of cases, one valid in the left hand store and one in the right. Finally, $\eta.\rho$ gives a relation between the types of these two cases. The final side-condition says this association is a *partial bijection*: a case on one side is associated to at most one case on the other side. Staging the relations and worlds is necessary due to a circularity here: a relation is (contravariantly) dependent on the current world, which contains relations. A relation in Rel_n is indexed by a World_n, but a World_n contains relations in $\mathrm{Rel}_{w.j}$, and $w.j < n$. In particular, $\mathrm{World}_0 = \emptyset$, so the definition is well-founded.

The next portion of the figure contains the definition of *world* extension $w' \sqsupseteq w$[2], representing the idea that w' is a possible "future" of w: the step index j is smaller and the states of the two sides have increased the number of allocated cases, but the old invariants are still there. We define strict extension $w' \sqsupset w$ to mean extension where the step has gotten strictly smaller. This allows us to define the *later* relation $\triangleright R$ which is used to break circular definitions in the logical relation. Next, we define our notion of non-indexed relation Rel_ω, which is what we quantify over in the interpretation of $\forall^v, \exists^v$. Then we define the restriction of interpretations and relations to a stage n. An infinitary relation can be "clamped" to any stage n using $\lfloor R \rfloor_n$. Finally, we define when two cases are related in an interpretation as $\eta \vDash (\sigma_l, \sigma_r, R)$.

Definition 225. We say γ, δ are valid instantiations of $\Gamma^{\sqsubseteq}$ in CBPV, written $(\gamma, \delta) \vDash \Gamma^{\sqsubseteq}$. when

- For each $i \in \{l, r\}$, there exists Σ_i such that for each $(x : A^{\sqsubseteq}) \in \Gamma^{\sqsubseteq}$, $\Sigma_i \mid \cdot \vdash \gamma_i(x) : [\![A_i]\!]$ when $\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$ and for each $X \in \Gamma^{\sqsubseteq}$, $\cdot \vdash \delta_i(X)$ and $\Sigma_i \mid \cdot \vdash \gamma_i(c_X) : \mathrm{Case}\ \delta_i(X)$.
- For each $X \in \Gamma^{\sqsubseteq}$, $\delta_R(X) \in \mathrm{Rel}_\omega[\delta_l(X), \delta_r(X)]$.

2 there is a clash of notation between precision $\sqsubseteq$ and world extension $\sqsupseteq$ but it should be clear which is meant at any time.

$$
\begin{array}{rcl}
\mathrm{Atom}_n[A_l, A_r] & = & \{(w, V_l, V_r) \mid w \in \mathrm{World}_n \wedge (w.\Sigma_l \mid \cdot \vdash V_l : A_l) \wedge (w.\Sigma_r \mid \cdot \vdash V_r : A_r)\} \\
\mathrm{CAtom}_n[A_l, A_r] & = & \{(w, V_l, V_r) \mid w \in \mathrm{World}_n \wedge w.\Sigma_l \mid \cdot \vdash M_l : FA_l \wedge w.\Sigma_r \mid \cdot \vdash M_r : FA_r\} \\
\mathrm{Rel}_n[A_l, A_r] & = & \{R \subseteq \mathrm{Atom}_n[A_l, A_r] \mid \forall (w, V_l, V_r) \in R, w' \sqsupseteq w.(w', V_l, V_r) \in R\} \\
\mathrm{World}_n & = & \{(j, \Sigma_l, \Sigma_r, \eta) \mid j < n \wedge \eta \in \mathrm{Interp}_j[\Sigma_l, \Sigma_r]\} \\
\mathrm{Interp}_n[\Sigma_l, \Sigma_r] & = & \{(size, f, \rho) \mid size \in \mathbb{N} \wedge f \in [size] \to ([\Sigma_l.size] \times [\Sigma_2.size]) \\
 & & \quad \wedge \rho : (i < size) \to Rel_n[\Sigma_l(f(i)); \Sigma_r(f(i))] \\
 & & \quad \wedge \forall i < j < size. f(i)_l \neq f(j)_l \wedge f(i)_r \neq f(j)_r\} \\
w' \sqsupset w. & = & (w' \sqsupseteq w) \wedge w'.j > w.j \\
w' \sqsupseteq w & = & w'.j \leq w.j \wedge w'.\Sigma_l \sqsupseteq w.\Sigma_l \wedge w'.\Sigma_r \sqsupseteq w.\Sigma_r \wedge w'.\eta \sqsupseteq \lfloor w.\eta \rfloor_{w'.j} \\
\Sigma' \sqsupseteq \Sigma & = & \Sigma'.size \geq \Sigma.size \wedge \forall i < \Sigma.size.\ \Sigma'(i) = \Sigma(i) \\
\eta' \sqsupseteq \eta & = & \eta'.size \geq \eta \wedge \forall i < \eta.size.\ \eta'.f(i) = \eta.f(i) \wedge \eta'.\rho(i) = \eta.\rho(i) \\
\triangleright R & = & \{(w, V_l, V_r) \mid \forall w' \sqsupset w.(w', V_l, V_r) \in R\} \\
\mathrm{Rel}_\omega[A_l, A_r] & = & \{R \subseteq \bigcup_{n \in \mathbb{N}} \mathrm{Atom}_n[A_l, A_r] \mid \forall n \in \mathbb{N}.\ \lfloor R \rfloor_n \in \mathrm{Rel}_n[A_l, A_r]\} \\
\lfloor \eta \rfloor_n & = & (\eta.size, \eta.f, \lambda i. \lfloor \rho(i) \rfloor_n) \\
\lfloor R \rfloor_n & = & \{(w, V_l, V_r) \mid w.j < n \wedge (w, V_l, V_r) \in R\} \\
\eta \vDash (\sigma_l, \sigma_r, R) & = & \exists i < \eta.size.\ \eta.f(i) = (\sigma_l, \sigma_r) \wedge \eta.\rho(i) = R\}
\end{array}
$$

Figure 10.27: Logical Relation Auxiliary Definitions

Definition 226. We define the extension of an interpretation η with a new association between seals as

$$\eta \boxplus (\sigma_l, \sigma_r, R) = (\eta.size + 1, (f, \eta.size \mapsto (\sigma_l, \sigma_r)), (\rho, \eta.size \mapsto R))$$

The top of Figure 10.28 contains the definition of the logical relation on values and computations. First, we write $\sim$ as a metavariable that ranges over two symbols: $\prec$ which indicates that we are counting steps on the left side, and $\succ$ which indicates we are counting steps on the right side. We then define the value relation $\mathcal{V}_n^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta \in \mathrm{Rel}_n[\delta_l(A_l), \delta_r(A_r)]$. Here γ maps the free term variables to pairs of values and δ maps free type variables to triples of two types and a relation between them. First, the definition for type variables looks up the relation in the relational substitution δ. Next, two values in ? are related when they are both injections into OSum, and the "payloads" of the injections are *later* related in the relation R which the world associates to the corresponding cases. The $\triangleright$ is used because we count pattern matching on OSum as a step. This also crucially lines up with the fact that pattern matching on the open sum type is the only reduction that consumes a step in our operational semantics. Note that this is a generalization of the logical relation definition for a recursive sum type, where each injection corresponds to a case of the sum. Here since the sum type is open, we must look in the world to see what cases are allocated. Next, the $\mathrm{tag}_G(A^{\sqsubseteq})$ case relates values on the left at some type A_l and values on the right of type ?. The definition states that the dynamically typed value must be an injection using the tag given by G, and that the payload of that injection must be related to

V_l with the relation given by $A^{\sqsubseteq}$. This case splits into two because we are pattern matching on a value of the open sum type, and so in the $\succ$ case we must decrement because we are consuming a step on the right, whereas in the $\prec$ case we do not decrement because we are only counting steps on the left. In the $\forall^{\nu} X.B^{\sqsubseteq}$ case, two values are related when in any future world, and any relation $R \in \mathrm{Rel}_{\omega}[A_l, A_r]$, and any pair of cases σ_l, σ_r that have $\lfloor R \rfloor_{w'.j}$ as their associated relation, if the values are instantiated with A_l, σ_l and A_r, σ_r respectively, then they behave as related computations. The intuition is that values of type $\forall^{\nu} X.B$ are parameterized by a type A and a tag for that type σ, but the relational interpretation of the two must be the *same*. This is the key to proving the $\mathtt{seal}_X$ and $\mathtt{unseal}_X$ cases of graduality. The fresh existential is dual in the choice of relation, but the same in its treatment of the case σ.

Next, we define the relation on expressions. The two expression relations, $\mathcal{E}^{\prec}[\![A^{\sqsubseteq}]\!]$ and $\mathcal{E}^{\succ}[\![A^{\sqsubseteq}]\!]$ capture the semantic idea behind graduality: either the left expression raises an error, or the two programs have the same behavior: diverge or return related values in $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]$. However, to account for step-indexing, each is an *approximation* to this notion where $\mathcal{E}^{\prec}[\![A^{\sqsubseteq}]\!]$ times out if the left side consumes all of the available steps $w.j$ (where $(\Sigma, M) \mapsto^{j}$ is shorthand for saying it steps to something in j steps), and $\mathcal{E}^{\succ}[\![A^{\sqsubseteq}]\!]$ times out if the right side consumes all of the available steps. relation is that when We define the *infinitary* version of the relations $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]$ and $\mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]$ a the union of all of the level n approximations.

Next, we give the relational interpretation of environments. The interpretation of the empty environment are empty substitutions with a valid world w. Extending with a value variable $x : A^{\sqsubseteq}$ means extending γ with a pair of values related by $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]$. For an abstract type variable X, first δ is extended with a pair of types and a relation between them. Then, γ must also be extended with a pair of *cases* encoding how these types are injected into the dynamic type. Crucially, just as with the $\forall^{\nu}, \exists^{\nu}$ value relations, these cases must be associated by w to the $w.j$ approximation of the *same* relation with which we extend δ. The interpretation of the *known* type variables $X \cong A^{\sqsubseteq}$ has the same basic structure, the key difference is that rather than using an arbitrary, δ is extended with the value relation $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]$.

With all of that preparation finished, we finally define the semantic interpretation of the graduality judgment $\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq}$ in the bottom of Figure 10.28. First, it says that both $M_l \sqsubseteq_{\prec} M_r$ and $M_l \sqsubseteq_{\succ} M_r$ hold, where we define $\sqsubseteq_{\sim}$ to mean that for any valid instantiation of the environments (including the preamble Γ_p), we get related computations. We can then define the "logical" Graduality theorem, that syntactic term precision implies semantic term precision, briefly, $\vdash$ implies $\vDash$.

$$
\begin{array}{rcl}
\mathcal{V}_n^{\sim}[\![X]\!]\gamma\delta & = & \lfloor \delta(X) \rfloor_n \\
\mathcal{V}_n^{\sim}[\![?]\!]\gamma\delta & = & \{(w, \mathtt{inj}_{\sigma_l}\ V_l, \mathtt{inj}_{\sigma_r}\ V_r) \in \mathrm{Atom}_n[?]\delta \mid w.\eta \vDash (\sigma_l, \sigma_r, R) \wedge (w, V_l, V_r) \in \triangleright R\} \\
\mathcal{V}_n^{\prec}[\![\mathrm{tag}_G(A^{\sqsubseteq})]\!]\gamma\delta & = & \{(w, V_l, \mathtt{inj}_{\gamma_r(\mathrm{case}(G))}\ V_r) \in \mathrm{Atom}_n[\mathrm{tag}_G(A^{\sqsubseteq})]\delta \mid (w, V_l, V_r) \in \mathcal{V}_n^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta\} \\
\mathcal{V}_n^{\succ}[\![\mathrm{tag}_G(A^{\sqsubseteq})]\!]\gamma\delta & = & \{(w, V_l, \mathtt{inj}_{\gamma_r(\mathrm{case}(G))}\ V_r) \in \mathrm{Atom}_n[\mathrm{tag}_G(A^{\sqsubseteq})]\delta \mid (w, V_l, V_r) \in (\triangleright \mathcal{V}^{\succ}[\![A^{\sqsubseteq}]\!]\gamma\delta)\} \\
\mathcal{V}_n^{\sim}[\![\mathtt{Bool}]\!]\gamma\delta & = & \{(w, \mathtt{true}, \mathtt{true}) \in \mathrm{Atom}_n[\mathtt{Bool}]\delta\} \cup \{(w, \mathtt{false}, \mathtt{false}) \in \mathrm{Atom}_n[\mathtt{Bool}]\delta\} \\
\mathcal{V}_n^{\sim}[\![A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}]\!]\gamma\delta & = & \{((V_{l1}, V_{l2}), (V_{r1}, V_{r2})) \in \mathrm{Atom}_n[A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}]\delta \\
 & & \mid (w, V_{l1}, V_{r1}) \in \mathcal{V}_n^{\sim}[\![A_1^{\sqsubseteq}]\!]\gamma\delta \wedge (w, V_{l2}, V_{r2}) \in \mathcal{V}_n^{\sim}[\![A_2^{\sqsubseteq}]\!]\gamma\delta\} \\
\mathcal{V}_n^{\sim}[\![A^{\sqsubseteq} \to B^{\sqsubseteq}]\!]\gamma\delta & = & \{(w, V_l, V_r) \in \mathrm{Atom}_n[A^{\sqsubseteq} \to B^{\sqsubseteq}]\delta \mid \forall w' \sqsupseteq w.(w', V_l', V_r') \in \mathcal{V}_n^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta. \\
 & & \quad (w', \mathtt{force}\ V_l\ V_l', \mathtt{force}\ V_r\ V_r') \in \mathcal{E}_n^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta\} \\
\mathcal{V}_n^{\sim}[\![\forall^{\nu} X.B^{\sqsubseteq}]\!]\gamma\delta & = & \{(w, V_l, V_r) \in \mathrm{Atom}_n[\forall^{\nu} X.A^{\sqsubseteq}]\delta \mid \\
 & & \quad \forall R \in \mathrm{Rel}_{\omega}[A_l, A_r].\forall w' \sqsupseteq w.\forall \sigma_l, \sigma_r.w'.\eta \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j}) \implies \\
 & & \quad (w', \mathtt{force}\ V_l\ [A_l]\ \sigma_l, \mathtt{force}\ V_r\ [A_r]\ \sigma_r) \in \mathcal{E}_n^{\sim}[\![B^{\sqsubseteq}]\!]\gamma'\delta' \\
 & & \quad (\text{where } \gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r), \text{ and } \delta' = \delta, X \mapsto (A_l, A_r, R))\} \\
\mathcal{V}_n^{\sim}[\![\exists^{\nu} X.B^{\sqsubseteq}]\!]\gamma\delta & = & \{(w, \mathtt{pack}\ (A_l, V_l), \mathtt{pack}\ (A_r, V_r)) \in \mathrm{Atom}_n[\exists^{\nu} X.B^{\sqsubseteq}]\delta \mid \\
 & & \quad \exists R \in \mathrm{Rel}_{\omega}[A_l, A_r].\forall w' \sqsupseteq w.\forall \sigma_l, \sigma_r.w'.\eta \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j}) \implies \\
 & & \quad (\mathtt{force}\ V_l\ \sigma_l, \mathtt{force}\ V_r\ \sigma_r) \in \mathcal{E}_n^{\sim}[\![B^{\sqsubseteq}]\!]\gamma'\delta' \\
 & & \quad (\text{where } \gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r), \text{ and } \delta' = \delta, X \mapsto (A_l, A_r, R))\} \\
\mathcal{E}_n^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta & = & \{(w, M_l, M_r) \in \mathrm{CAtom}_n[A^{\sqsubseteq}]\delta \mid (w.\Sigma_l, M_l) \mapsto^{w.j} \vee ((w.\Sigma_l, M_l) \mapsto^{<w.j} (\Sigma_l', \mathcal{U})) \\
 & & \vee (\exists w' \sqsupseteq w.(w', V_l, V_r) \in \mathcal{V}_n^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta. \\
 & & \quad (w.\Sigma_l, M_l) \mapsto^{w'.j - w.j} (w'.\Sigma_l, \mathtt{ret}\ V_l) \wedge (w.\Sigma_r, M_r) \mapsto^{*} (w'.\Sigma_r, \mathtt{ret}\ V_r))\} \\
\mathcal{E}_n^{\succ}[\![A^{\sqsubseteq}]\!]\gamma\delta & = & \{(w, M_l, M_r) \in \mathrm{CAtom}_n[A^{\sqsubseteq}]\delta \mid (w.\Sigma_r, M_r) \mapsto^{w.j} \vee ((w.\Sigma_l, M_l) \mapsto^{*} (\Sigma_l', \mathcal{U})) \\
 & & \vee \exists w' \sqsupseteq w.(w', V_l, V_r) \in \mathcal{V}_n^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta. \\
 & & \quad (w.\Sigma_l, M_l) \mapsto^{*} (w'.\Sigma_l, \mathtt{ret}\ V_l) \wedge (w.\Sigma_r, M_r) \mapsto^{w'.j - w.j} (w'.\Sigma_r, \mathtt{ret}\ V_r)\}
\end{array}
$$

$$\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta = \bigcup_{n \in \mathbb{N}} \mathcal{V}_n^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta \qquad \mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta = \bigcup_{n \in \mathbb{N}} \mathcal{E}_n^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$$

$$
\begin{array}{rcl}
\mathcal{G}^{\sim}[\![\cdot]\!] & = & \{(w, \emptyset, \emptyset) \mid \exists n.w \in \mathrm{World}_n\} \\
\mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}, x : A^{\sqsubseteq}]\!] & = & \{(w, (\gamma, x \mapsto (V_l, V_r)), \delta) \mid (w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!] \wedge (w, V_l, V_r) \in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta\} \\
\mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}, X]\!] & = & \{(w, (\gamma, c_X \mapsto (\sigma_l, \sigma_r)), \delta, X \mapsto (A_l, A_r, R)) \mid (w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!] \\
 & & \quad \wedge R \in \mathrm{Rel}_{\omega}[A_l, A_r] \wedge (\sigma_l, \sigma_r, \lfloor R \rfloor_{w.j}) \in w\} \\
\mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}, X \cong A^{\sqsubseteq}]\!] & = & \{(w, (\gamma, c_X \mapsto (\sigma_l, \sigma_r)), \delta, X \mapsto (A_l, A_r, \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta)) \mid (w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!] \\
 & & \quad w \vDash (\sigma_l, \sigma_r, \mathcal{V}_{w.j}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta)\}
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq} & = & \Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq_{\prec} M_r \in A^{\sqsubseteq} \wedge \Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq_{\succ} M_r \in A^{\sqsubseteq} \\
\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq_{\sim} M_r \in A^{\sqsubseteq} & = & \forall (w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_p, \Gamma^{\sqsubseteq}]\!]. \\
 & & \quad (w, [\![M_l]\!][\gamma_l][\delta_l], [\![M_r]\!][\gamma_l][\delta_l]) \in \mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta
\end{array}
$$

Figure 10.28: Graduality/Parametricity Logical Relation

Theorem 227 (Logical Graduality). *If* $\Gamma^{\sqsubseteq} \vdash M_l \sqsubseteq M_r : A^{\sqsubseteq}$, *then* $\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq}$

The proof is by induction on the term precision derivation. Each case is proven as a separate lemma. The cases of $\forall^v, \exists^v$, sealing and unsealing follow because the treatment of type variables between the value and environment relations is the same. The cast cases are the most involved, following by two lemmas proven by induction over precision derivations: one for when the cast is on the left, and the other when the cast is on the right.

Finally, we prove the operational graduality theorem as a corollary of the logical graduality Theorem 227 and the adequacy Theorem 217. By constructing a suitable starting world w_{pre} that allocates the globally known tags, we ensure the operational graduality property holds for the code translated to $\text{CBPV}_{\text{OSum}}$, and then the simulation theorem implies the analogous property holds for the PolyC^v operational semantics.

10.6.3.1 *Logical Lemmas*

Logical Lemmas

Lemma 228. *If* $R \in Rel_\omega[A_l, A_r]$, *then* $\lfloor R \rfloor_n \in Rel_n[A_l, A_r]$

Proof. Direct by definition. □

Lemma 229. $\lfloor \mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta \rfloor_n = \mathcal{V}_n^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta$

Proof. Direct by definition. □

Lemma 230. *If* $R \in Rel_n[A_l, A_r]$, *then* $\triangleright R \in Rel_n[A_l, A_r]$

Proof. If $(w, V_l, V_r) \in (\triangleright R)$ and $w' \sqsupseteq w$, then to show $(w', V_l, V_r) \in \triangleright R$, we need to show that for any $w'' \sqsupset w'$, that $(w'', V_l, V_r) \in R$, but this follows because $(w, V_l, V_r) \in \triangleright R$ and $w'' \sqsupset w' \sqsupseteq w$. □

Lemma 231. *Let* $\Gamma^{\sqsubseteq}$ *be a well-formed context, with* $\Gamma^{\sqsubseteq} \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$. *If* $(\gamma, \delta) \vDash \Gamma^{\sqsubseteq}$, *then* $\mathcal{V}_n^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta \in Rel_n[A_l, A_r]$.

Proof. By induction on $A^{\sqsubseteq}$.

1. X: by Lemma 228.
2. ?: If $(w, \texttt{inj}_{\sigma_l}\ V_l, \texttt{inj}_{\sigma_r}\ V_r) \in \mathcal{V}_n^\sim[\![?]\!]\gamma\delta$ and $w' \sqsupseteq w$, then there exists $R \in \text{Rel}_n[A_l, A_r]$ with $w.\eta \vDash (\sigma_l, \sigma_r, R)$ and $(w, V_l, V_r) \in \triangleright R$. By definition of $\sqsupseteq$, we have that $w' \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$ so it is sufficient to show $(w, V_l, V_r) \in \triangleright \lfloor R \rfloor_{w'.j}$, which follows by Lemma 230 that later preserves monotonicity.
3. $\text{tag}_G(A^{\sqsubseteq})$: by inductive hypothesis, using Lemma 230 in the $\succ$ case.

4. `Bool`: immediate
5. $\times$: immediate by inductive hypothesis.
6. $\to$: If $(w, V_l, V_r) \in \mathcal{V}_n^{\sim}[\![A^{\sqsubseteq} \to B^{\sqsubseteq}]\!]\gamma\delta$ and $w' \sqsupseteq w$. Then given $w'' \sqsupseteq w'$ and $(w'', V_l', V_r') \in \mathcal{V}_n^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$, we need to show

   $$(w'', \texttt{force } V_l\, V_l', \texttt{force } V_r\, V_r') \mathcal{E}_n^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta,$$

   which this holds by relatedness of V_l, V_r because $w'' \sqsupseteq w$ by transitivity of world extension.
7. $\forall^{\nu}$, $\exists^{\nu}$: similar to the $\to$ case.

□

Lemma 232. *If $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!]$ and $w' \sqsupseteq w$, then $(w', \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!]$.*

Proof. By induction on $\Gamma^{\sqsubseteq}$, uses monotonicity of $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]$ □

Corollary 233. $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta \in Rel_{\omega}[A_l, A_r]$

Lemma 234 (Anti-Reduction)**.** *1. If $w' \sqsupseteq w$ and $(w.\Sigma_l, M_l) \mapsto^{w.j - w'.j} (w'.\Sigma_l, M_l')$ and $(w.\Sigma_r, M_r) \mapsto^{*} (w'.\Sigma_r, M_r')$ and $(w', M_l', M_r') \in \mathcal{E}^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta$, then $(w, M_l, M_r) \in \mathcal{E}^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta$.*

2. If $w' \sqsupseteq w$ and $(w.\Sigma_l, M_l) \mapsto^{w.j - w'.j} (w'.\Sigma_l, M_l')$ and $(w.\Sigma_r, M_r) \mapsto^{*} (w'.\Sigma_r, M_r')$ and $(w', M_l', M_r') \in \mathcal{E}^{\succ}[\![A^{\sqsubseteq}]\!]\gamma\delta$, then $(w, M_l, M_r) \in \mathcal{E}^{\succ}[\![A^{\sqsubseteq}]\!]\gamma\delta$.

Proof. We do the $\prec$ case, the other is symmetric. By case analysis on $(w', M_l', M_r') \in \mathcal{E}^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta$.

1. If $w'.\Sigma_l, M_l' \mapsto^{w'.j+1}$, then $(w.\Sigma_l, M_l) \mapsto^{w.j - w'.j + w'.j + 1}$ and $w.j - w'.j + w'.j + 1 = w.j + 1$.
2. If $w'.\Sigma_l, M_l' \mapsto^{j} \Sigma_l', \upsilon$, with $j \leq w.j$, then $w.\Sigma_l, M_l \mapsto^{w.j - w'.j + j} \Sigma_l', \upsilon$ and $w.j - w'.j + j \leq w.j$ since $j - w'.j \leq 0$.
3. Finally, if there is some $w'' \sqsupseteq w'$ and $(w'', V_l, V_r) \in \mathcal{V}^{\prec}[\![A^{\sqsubseteq}]\!]\gamma\delta$ with $w'.\Sigma_l, M_l' \mapsto^{w'j. - w''.j} w''.\Sigma_l, \texttt{ret } V_l$ and $w'.\Sigma_r, M_r' \mapsto^{*} w'', \Sigma_r, \texttt{ret } V_r$, then $w.\Sigma_l, M_l \mapsto^{w.j - w'.j + w'.j - w''.j} w''.\Sigma_l, \texttt{ret } V_l''$ and $w.\Sigma_r, M_r \mapsto^{*} w'', \Sigma_r, \texttt{ret } V_r$ and $w.j - w'.j + w'.j - w''.j = w.j - w''.j$ so the result holds.

□

Lemma 235 (Pure Anti-Reduction)**.** *If $(w, M_l', M_r') \in \mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$ and $(w.\Sigma_l, M_l) \mapsto^{0} (w.\Sigma_l, M_l')$ and $(w.\Sigma_r, M_r) \mapsto^{0} (w.\Sigma_r, M_r')$, then $(w, M_l, M_r) \in \mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$.*

Proof. Immediate corollary of anti-reduction Lemma 234 □

Lemma 236 (Pure Forward Reduction). *If* $(w, M_l', M_r') \in \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$ *and* $(w.\Sigma_l, M_l) \mapsto^0 (w.\Sigma_l, M_l')$ *and* $(w.\Sigma_r, M_r) \mapsto^0 (w.\Sigma_r, M_r')$*, then* $(w, M_l, M_r) \in \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$*.*

Proof. By determinism of evaluation. □

Lemma 237 (Monadic bind). *If* $(w, M_l, M_r) \in \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$ *and for all* $w' \sqsupseteq w$*, and* $(w', V_l, V_r) \in \mathcal{V}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$*,* $(w', S_l[\texttt{ret } V_l], S_r[\texttt{ret } V_r]) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta$*, then* $(w, S_l[M_l], S_r[M_r]) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta$*.*

Proof. We show the proof for $\mathcal{E}^\prec[\![A^\sqsubseteq]\!]$, the $\succ$ case is symmetric. By case analysis on $(w, M_l, M_r) \in \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$.

1. If $w.\Sigma_l, M_l \mapsto^{w.j+1}$, then $w.\Sigma_l, S[M_l] \mapsto^{w.j+1}$.
2. If $w.\Sigma_l, M_l \mapsto^j w.\Sigma_l', \mho$, then $w.\Sigma_l, S[M_l] \mapsto^j w.\Sigma_l', \mho$.
3. Otherwise there exists w' and $(w', V_l, V_r) \in \mathcal{V}^\prec[\![A^\sqsubseteq]\!]\gamma\delta$ with $w.\Sigma_l, M_l \mapsto^{w.j-w'.j} w'.\Sigma_l, \texttt{ret } V_l$ and $w.\Sigma_r, M_r \mapsto^* w'.\Sigma_r, \texttt{ret } V_r$. Then $w.\Sigma_l, S_l[M_l] \mapsto^{w.j-w'.j} w'.\Sigma_l, S[\texttt{ret } V_l]$ and $w.\Sigma_r, S_r[M_r] \mapsto^* w'.\Sigma_r, S_r[\texttt{ret } V_r]$, and the result follows by the assumption.

□

Pure evaluation is monotone.

Lemma 238. *If* $\Sigma, M \mapsto^* \Sigma, N$*, then for any* $\Sigma' \sqsupseteq \Sigma$*,* $\Sigma', M \mapsto^* \Sigma', N$*.*

Clamping

Lemma 239. *If* $(w, V_l, V_r) \in R$ *and* $w.j \leq n$*, then* $(w, V_l, V_r) \in \lfloor R \rfloor_n$*.*

Proof. Direct from definition. □

Tag-to-type

Lemma 240. $\mathcal{V}_n^\sim[\![G]\!]\gamma\delta = \lfloor \delta_R(case(G)) \rfloor_n$

Proof. Direct from definition □

Lemma 241 (Weakening). *If* $\Gamma^\sqsubseteq \vdash A^\sqsubseteq$ *and* $\Gamma^\sqsubseteq \subseteq \Gamma^{\sqsubseteq\prime}$ *and* $(w, \gamma, \delta) \in \mathcal{G}^\sim[\![\Gamma^\sqsubseteq]\!]$ *and* $(w, \gamma', \delta') \in \mathcal{G}^\sim[\![\Gamma^{\sqsubseteq\prime}]\!]$*, where* $\gamma \subseteq \gamma'$ *and* $\delta \subseteq \delta'$*, then all of the following are true:*

$$\mathcal{V}^\sim[\![A^\sqsubseteq]\!]\gamma\delta = \mathcal{V}^\sim[\![A^\sqsubseteq]\!]\gamma'\delta'$$

$$\mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta = \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma'\delta'$$

Proof. Straightforward, by induction over $\Gamma^{\sqsubseteq\prime}$. □

10.6.3.2 *Congruence Cases*

To prove the cast left lemma, we need the following lemma that casts always either error or terminate with a value on well-typed inputs.

Lemma 242 (Casts don't diverge)**.** *If* $\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$*, then for any* Σ*, and* $\Sigma \mid \cdot \vdash \gamma : [\![\Gamma]\!]$*,*

1. *If* $\Sigma \mid \cdot \vdash V : A_l$*, then either* $\Sigma, [\![\langle A^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_l][\gamma] \mapsto^* \Sigma, \mathcal{U}$ *or* $\Sigma, [\![\langle A^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_l][\gamma] \mapsto^* \Sigma, \mathtt{ret}\ V'$*.*
2. *If* $\Sigma \mid \cdot \vdash V : A_r$*, then either* $\Sigma, [\![\langle A^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V][\gamma] \mapsto^* \Sigma, \mathcal{U}$ *or* $\Sigma, [\![\langle A^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V][\gamma] \mapsto^* \Sigma, \mathtt{ret}\ V'$*.*

Proof. By induction on $A^{\sqsubseteq}$.

1. If $A^{\sqsubseteq} \in \{?, \mathtt{Bool}\}$ the cast is trivial.
2. Case $A^{\sqsubseteq} = \mathrm{tag}_G(AG^{\sqsubseteq})$:
   a) The upcast definition expands as follows:

$$[\![\langle \mathrm{tag}_G(AG^{\sqsubseteq})\rangle\!\updownarrow]\!][\mathtt{ret}\ V][\gamma] = x \leftarrow [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V][\gamma]; \mathtt{ret}\ \mathtt{inj}_\sigma\ x$$

   where $\sigma = \gamma(\mathrm{case}(G))$. By inductive hypothesis, $[\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V][\gamma]$ either errors (in which case the whole term errors), or runs to a value V', in which case

$$x \leftarrow \mathtt{ret}\ V'; \mathtt{ret}\ \mathtt{inj}_\sigma\ x \mapsto^* \mathtt{ret}\ \mathtt{inj}_\sigma\ V'$$

   b) The downcast definition expands as follows:

$$[\![\langle \mathrm{tag}_G(AG^{\sqsubseteq})\rangle\!\downarrow]\!][\mathtt{ret}\ V][\gamma] = x \leftarrow \mathtt{ret}\ V; \mathtt{match}\ x\ \mathtt{with}\ \sigma\{\mathtt{inj}\ y.[\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ y][\gamma] \mid \mathcal{U}\}$$

   Since $\Sigma \mid \cdot V : \mathrm{OSum}$, $V = \mathtt{inj}_{\sigma'}\ V'$ for some $\sigma' \in \Sigma$.
   i. If $\sigma' = \sigma$, then

$$\mathtt{match}\ \mathtt{inj}_\sigma\ V'\ \mathtt{with}\ \sigma\{\mathtt{inj}\ y.[\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ y][\gamma] \mid \mathcal{U}\} \mapsto^1 [\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V'][\gamma]$$

   and then it follows by inductive hypothesis with $AG^{\sqsubseteq}$.
   ii. If $\sigma' \neq \sigma$, then

$$\mathtt{match}\ \mathtt{inj}_\sigma\ V'\ \mathtt{with}\ \sigma\{\mathtt{inj}\ y.[\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ y][\gamma] \mid \mathcal{U}\} \mapsto^1 \mathcal{U}$$

   and the result holds.

3. If $A^{\sqsubseteq} = A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}$, we consider the downcast case, the upcast is entirely symmetric. First,

$$\begin{aligned}[\![\langle A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V][\gamma] = x \leftarrow\ & \mathtt{ret}\ V; \\ & \mathtt{let}\ (x_1, x_2) = x; \\ & y_1 \leftarrow [\![\langle A_1^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ x_1][\gamma]; \\ & y_2 \leftarrow [\![\langle A_2^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ x_2][\gamma]; \\ & \mathtt{ret}\ (y_1, y_2)\end{aligned}$$

Next, since V is well-typed, $V = (V_1, V_2)$. Then,

$$\begin{array}{ll} x \leftarrow \texttt{ret}\ V; & \mapsto^0 y_1 \leftarrow [\![\langle A_1^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_1][\gamma]; \\ \texttt{let}\ (x_1, x_2) = x; & \quad y_2 \leftarrow [\![\langle A_2^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_2][\gamma]; \\ y_1 \leftarrow [\![\langle A_1^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ x_1][\gamma]; & \quad \texttt{ret}\ (y_1, y_2) \\ y_2 \leftarrow [\![\langle A_2^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ x_2][\gamma]; & \\ \texttt{ret}\ (y_1, y_2) & \end{array}$$

Applying the inductive hypothesis to $A_1^{\sqsubseteq}$, either $[\![\langle A_1^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_1][\gamma]$ errors (in which case the whole term errors), or it runs to a value V_1'. Then we need to show

$$\begin{array}{l} y_2 \leftarrow [\![\langle A_2^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_2][\gamma]; \\ \texttt{ret}\ (V_1', y_2) \end{array}$$

errors or terminates. Applying the inductive hypothesis to $A_2^{\sqsubseteq}$, either $[\![\langle A_2^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_2][\gamma]$ errors (in which case the whole term errors), or it runs to a value V_2'. Then the whole term runs to $\texttt{ret}\ (V_1', V_2')$.

4. If $A^{\sqsubseteq} = A_i^{\sqsubseteq} \to A_o^{\sqsubseteq}$, we consider the downcast case (upcast is symmetric). The downcast definition expands as follows:

$$\begin{array}{l} [\![\langle A_i^{\sqsubseteq} \to A_o^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V][\gamma] = f \leftarrow \texttt{ret}\ V; \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{ret thunk}\ \lambda x. \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad [\![\langle A_o^{\sqsubseteq}\rangle\downarrow]\!][y \leftarrow [\![\langle A_i^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ x][\gamma]; \texttt{force}\ f\ y][\gamma] \end{array}$$

   Which steps immediately to a value.

5. If $A^{\sqsubseteq} = \forall^{\nu} X.A_o^{\sqsubseteq}$, then it follows by similar reasoning to the function case, that is, it immediately terminates.

6. If $A^{\sqsubseteq} = \exists^{\nu} X.A_o^{\sqsubseteq}$, we consider the downcast case (upcast is symmetric). The definition expands as follows:

$$[\![\langle \exists^{\nu} X.A_o^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V][\gamma] = \texttt{unpack}\ (X, x) = \texttt{ret}\ V; \texttt{ret thunk}\ \lambda c_X.[\![\langle A_o^{\sqsubseteq}\rangle\downarrow]\!][\texttt{force}\ x\, c_X]$$

   Which steps immediately to a value.

□

Lemma 243 (Cast Right)**.** *For any* $\Gamma^{\sqsubseteq}$*, if* $\Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C$ *and* $\Gamma^{\sqsubseteq} \vdash AB^{\sqsubseteq} : A \sqsubseteq B$*, and* $\Gamma' \vdash BC^{\sqsubseteq} : B \sqsubseteq C$*, Then if* $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!]$*,*

1. *If* $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![AB^{\sqsubseteq}]\!]\gamma\delta$*, then* $(w, \texttt{ret}\ V_l, [\![\langle BC^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ V_r][\gamma_r]) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$

2. *If* $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$*, then* $(w, \texttt{ret}\ V_l, [\![\langle BC^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_r][\gamma_r]) \in \mathcal{E}^{\sim}[\![AB^{\sqsubseteq}]\!]\gamma\delta$

Proof. By induction on $BC^{\sqsubseteq}$.

1. If $BC^{\sqsubseteq} \in \{\texttt{Bool}, ?, X\}$, then the cast is trivial.
2. If $BC^{\sqsubseteq} = \mathrm{tag}_G(BG^{\sqsubseteq})$, then $AC^{\sqsubseteq} = \mathrm{tag}_G(AG^{\sqsubseteq})$.
   a) For the upcast case, we are given that $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![AB^{\sqsubseteq}]\!]\gamma\delta$ and we need to prove that

$$(w, \texttt{ret}\ V_l, y \leftarrow [\![\langle BG^{\sqsubseteq}\rangle\!\updownarrow]\!][V_r][\gamma_r]; \texttt{ret}\ \texttt{inj}_G\ y) \in \mathcal{E}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

   By inductive hypothesis, we know

$$(w, \texttt{ret}\ V_l, [\![\langle BG^{\sqsubseteq}\rangle\!\updownarrow]\!]V_r[\gamma_r]) \in \mathcal{E}^{\sim}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

   We then use monadic bind (Lemma 237). Suppose $w' \sqsupseteq w$ and $(w', V_l', V_r') \in \mathcal{V}^{\sim}[\![AG^{\sqsubseteq}]\!]\gamma\delta$. We need to show that

$$(w', \texttt{ret}\ V_l', y \leftarrow \texttt{ret}\ V_r'; \texttt{ret}\ \texttt{inj}_G\ y) \in \mathcal{E}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

   By anti reduction, it is sufficient to show

$$(w', V_l, \texttt{inj}_{\gamma_r(G)}\ V_r) \in \mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

   i. If $\sim = \prec$, we need to show $(w', V_l, V_r) \in \mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$, which follows by inductive hypothesis.
   ii. If $\sim = \succ$, we need to show $(w', V_l, V_r) \in \triangleright\mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$, that is for any $w'' \sqsupset w'$, $(w'', V_l, V_r) \in \mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$ which also follows by inductive hypothesis.

   b) For the downcast case, we know $(V_l, V_r) \in \mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]w\gamma$. Let $\sigma_r = \gamma_r(\mathrm{case}(G))$.
   i. In the $\prec$ case, we know $V_r = \texttt{inj}_{\sigma_r}\ V_r'$ and

$$(w, V_l, V_r') \in \mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

   we need to show

$$(w, \texttt{ret}\ V_l, \texttt{match}\ (\texttt{inj}_{\sigma_r}\ V_r')\ \texttt{with}\ \sigma_r\{\texttt{inj}\ x.[\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][x][\gamma_r] \mid \mho\}) \in \mathcal{E}^{\prec}[\![AB^{\sqsubseteq}]\!]\gamma\delta$$

   the right hand side reduces to $[\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][V'']$, so by anti-reduction it is sufficient to show

$$(w, \texttt{ret}\ V_l, [\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][V_r']) \in \mathcal{E}^{\prec}[\![AB^{\sqsubseteq}]\!]\gamma\delta$$

   which follows by inductive hypothesis.
   ii. In the $\succ$ case, we know $V_r = \texttt{inj}_{\sigma_r}\ V_r'$ and

$$(w, V, V_r') \in \triangleright(\mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta)$$

   (note the $\triangleright$). We need to show

$$(w, \texttt{ret}\ V_l, \texttt{match}\ (\texttt{inj}_{\sigma_r}\ V_r')\ \texttt{with}\ \sigma_r\{\texttt{inj}\ x.[\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][x][\gamma_r] \mid \mho\}) \in \mathcal{E}^{\succ}[\![AB^{\sqsubseteq}]\!]\gamma\delta$$

   the right hand side takes 1 step to $[\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][V_r']$.

A. If $w.j = 0$, then we are done.

B. Otherwise, define $w' = (w.j - 1, w.\Sigma_l, w.\Sigma_r, \lfloor w.\eta \rfloor_{w.j-1})$. Then by anti-reduction, it is sufficient to show

$$(w', \mathtt{ret}\ V_l, [\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][V_r']) \in \mathcal{E}^{\succ}[\![AB^{\sqsubseteq}]\!]\gamma\delta$$

$$(w', \mathtt{ret}\ V_l, [\![\langle BG^{\sqsubseteq}\rangle\!\downarrow]\!][V_r']) \in \mathcal{E}^{\succ}[\![AB^{\sqsubseteq}]\!]$$

By inductive hypothesis, it is sufficient to show

$$(w', V_l, V_r') \in \mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

which follows by assumption because $w' \sqsupset w$.

3. If $BC^{\sqsubseteq} = BC_1^{\sqsubseteq} \times BC_2^{\sqsubseteq}$, then by precision inversion also $AC^{\sqsubseteq} = AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}$ and $AB^{\sqsubseteq} = AB_1^{\sqsubseteq} \times AB_2^{\sqsubseteq}$. We consider the upcast case, the downcast case follows by an entirely analogous argument.

   Given $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![BC_1^{\sqsubseteq} \times BC_2^{\sqsubseteq}]\!]\gamma\delta$, we need to show

   $$(w, \mathtt{ret}\ V_l, [\![\langle BC_1^{\sqsubseteq} \times BC_2^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_r]) \in \mathcal{E}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]\gamma\delta$$

   Expanding definitions, and applying anti-reduction , this reduces to showing

   $$\begin{array}{ll}(w, \mathtt{ret}\ V_l, \mathtt{let}\ (y_1, y_2) = V_r; &) \in \mathcal{E}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]\gamma\delta \\ \quad z_1 \leftarrow [\![\langle BC_1^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ y_1][\gamma_r]; & \\ \quad z_2 \leftarrow [\![\langle BC_2^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ y_2][\gamma_r]; & \\ \quad \mathtt{ret}\ (z_1, z_2) & \end{array}$$

   Since $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![BC_1^{\sqsubseteq} \times BC_2^{\sqsubseteq}]\!]\gamma\delta$, we know

   $$V_l = (V_{l1}, V_{l2}) \qquad V_r = (V_{r1}, V_{r2})$$

   $$(V_{l1}, V_{r1}) \in \mathcal{V}^{\sim}[\![BC_1^{\sqsubseteq}]\!]\gamma\delta \qquad (V_{l2}, V_{r2}) \in \mathcal{V}^{\sim}[\![BC_2^{\sqsubseteq}]\!]\gamma\delta$$

   So after a reduction we need to show

   $$\begin{array}{ll}(w, \mathtt{ret}\ V_l, z_1 \leftarrow [\![\langle BC_1^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_{r1}][\gamma_r];) & \in \mathcal{E}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]\gamma\delta \\ \quad z_2 \leftarrow [\![\langle BC_2^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_{r2}][\gamma_r]; & \\ \quad \mathtt{ret}\ (z_1, z_2) & \end{array}$$

   By forward reduction, it is sufficient to prove the following, (which is amenable to monadic bind):

$$\begin{array}{ll}(w, z \leftarrow \mathtt{ret}\ V_{l1}; z_1 \leftarrow [\![\langle BC_1^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_{r1}][\gamma_r];) & \in \mathcal{E}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]\gamma\delta w\gamma \\ \quad z_2 \leftarrow \mathtt{ret}\ V_{l2}; z_2 \leftarrow [\![\langle BC_2^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ V_{r2}][\gamma_r]; & \\ \quad \mathtt{ret}\ (z_1, z_2) \quad \mathtt{ret}\ (z_1, z_2) & \end{array}$$

We then apply monadic bind with the inductive hypothesis for $BC_1^{\sqsubseteq}$. Given $w' \sqsupseteq w$ and $(V'_{l1}, V'_{r1}) \in \mathcal{V}^{\sim}[\![AC_1^{\sqsubseteq}]\!]\gamma\delta$ the goal reduces to

$$\begin{array}{l}(w', z_2 \leftarrow \texttt{ret}\ V_{l2}; , z_2 \leftarrow [\![\langle BC_2^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_{r2}][\gamma_r];) \in \mathcal{E}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]\gamma\delta \\ \quad \texttt{ret}\ (V'_{l1}, z_2) \quad \texttt{ret}\ (V'_{r1}, z_2)\end{array}$$

We then apply another monadic bind with the inductive hypothesis for $BC_2^{\sqsubseteq}$. Given $w'' \sqsupseteq w'$ and $(V'_{l2}, V'_{r2}) \in \mathcal{V}^{\sim}[\![AC_2^{\sqsubseteq}]\!]w'\gamma$, the goal reduces to

$$(w'', (V'_{l1}, V'_{l2}), (V'_{r1}, V'_{r2})) \in \mathcal{V}^{\sim}[\![AC_1^{\sqsubseteq} \times AC_2^{\sqsubseteq}]\!]w''\gamma$$

which follows immediately by our assumptions from monadic bind.

4. If $BC^{\sqsubseteq} = BC_i^{\sqsubseteq} \to BC_o^{\sqsubseteq}$, then by precision inversion also $AC^{\sqsubseteq} = AC_i^{\sqsubseteq} \to AC_o^{\sqsubseteq}$ and $AB^{\sqsubseteq} = AB_i^{\sqsubseteq} \to AB_o^{\sqsubseteq}$. We consider the upcast case, the downcast case follows by an entirely analogous argument.

   Given $(V_l, V_r) \in \mathcal{V}^{\sim}[\![BC_i^{\sqsubseteq} \to BC_o^{\sqsubseteq}]\!]\gamma\delta$, we need to show

   $$(w, \texttt{ret}\ V_l, [\![\langle BC_i^{\sqsubseteq} \to BC_o^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ V_r][\gamma_r]) \in \mathcal{E}^{\sim}[\![AC_i^{\sqsubseteq} \to AC_o^{\sqsubseteq}]\!]\gamma\delta$$

   Expanding definitions, this reduces to showing

   $$\begin{array}{l}(w, V_l, \texttt{thunk}\ (\lambda x.y \leftarrow [\![\langle BC_i^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ x][\gamma_r];)) \in \mathcal{V}^{\sim}[\![AC_i^{\sqsubseteq} \to AC_o^{\sqsubseteq}]\!]\gamma\delta \\ \qquad\qquad z \leftarrow \texttt{force}\ V_r y; \\ \qquad\qquad [\![\langle BC_o^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ z][\gamma_r]\end{array}$$

   Let $w' \sqsupseteq w$ be a future world and $(w', V_{li}, V_{ri}) \in \mathcal{V}^{\sim}[\![AC_i^{\sqsubseteq}]\!]\gamma\delta$. Then our goal reduces to showing

   $$\begin{array}{l}(w', \texttt{force}\ V_l V_{li}, y \leftarrow [\![\langle BC_i^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_{ri}][\gamma_r];) \in \mathcal{E}^{\sim}[\![AC_o^{\sqsubseteq}]\!]\gamma\delta \\ \qquad\qquad z \leftarrow \texttt{force}\ V_r y; \\ \qquad\qquad [\![\langle BC_o^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ z][\gamma_r]\end{array}$$

   by forward reduction, it is sufficient to show

   $$\begin{array}{l}(w', y \leftarrow V_{li}; \ \ , y \leftarrow [\![\langle BC_i^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_{ri}][\gamma_r];) \in \mathcal{E}^{\sim}[\![AC_o^{\sqsubseteq}]\!]\gamma\delta \\ \quad \texttt{force}\ V_l y\ z \leftarrow \texttt{force}\ V_r y; \\ \qquad\qquad [\![\langle BC_o^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ z][\gamma_r]\end{array}$$

   We then use the inductive hypothesis on $BC_i^{\sqsubseteq}$ (which applies because of downward-closure) and monadic bind: assume $w'' \sqsupseteq w'$ and $(w'', V'_{li}, V'_{ri}) \in \mathcal{V}^{\sim}[\![AC_i^{\sqsubseteq}]\!]\gamma\delta$. We need to show

   $$\begin{array}{l}(w'', \texttt{force}\ V_l V'_{li}, z \leftarrow \texttt{force}\ V_r V'_{ri}; \quad) \in \mathcal{E}^{\sim}[\![AC_o^{\sqsubseteq}]\!]\gamma\delta \\ \qquad\qquad [\![\langle BC_o^{\sqsubseteq}\rangle\updownarrow]\!][\texttt{ret}\ z][\gamma_r]\end{array}$$

We apply monadic bind again, noting that the applications are related by assumption and downward closure. Assume $w''' \sqsupseteq w''$ and $(w''', V_{lo}, V_{ro}) \in \mathcal{V}^{\sim}[\![AB^{\sqsubseteq}_o]\!]\gamma\delta$. By anti-reduction , the goal reduces to showing

$$(\texttt{ret}\ V_o, [\![\langle BC^{\sqsubseteq}_o\rangle\!\updownarrow]\!][\texttt{ret}\ V'_o][\gamma]) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}_o]\!]\gamma\delta$$

which follows by inductive hypothesis for $BC^{\sqsubseteq}_o$.

5. If $BC^{\sqsubseteq} = \forall^{\nu} X.BC^{\sqsubseteq}_o$, then by precision inversion also $AC^{\sqsubseteq} = \forall^{\nu} X.AC^{\sqsubseteq}_o$ and $AB^{\sqsubseteq} = \forall^{\nu} X.AB^{\sqsubseteq}_o$. We consider the upcast case, the downcast case follows by an entirely analogous argument. Given $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![\forall^{\nu} X.BC^{\sqsubseteq}_o]\!]\gamma\delta$, we need to show

$$(w, \texttt{ret}\ V_l, [\![\langle \forall^{\nu} X.BC^{\sqsubseteq}_o\rangle\!\updownarrow]\!][\texttt{ret}\ V_r][\gamma_r]) \in \mathcal{E}^{\sim}[\![\forall^{\nu} AC^{\sqsubseteq}_o]\!]\gamma\delta$$

Expanding definitions and applying anti-reduction, this reduces to showing

$$(w, V_l, V'_r) \in \mathcal{V}^{\sim}[\![\forall^{\nu} AC^{\sqsubseteq}_o]\!]\gamma\delta$$

where

$$V'_r = \texttt{thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.[\![\langle BC^{\sqsubseteq}_o\rangle\!\downarrow]\!][(\texttt{force}\ V_r)\ X\ c_X][\gamma_r])$$

Let $w' \sqsupseteq w$, $R \in Rel[A_l, A_r]$, and $w.\eta \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w.j})$, then we need to show that

$$(w', (\texttt{force}\ V_l)\ A_l\ \sigma_l, (\texttt{force}\ V'_r)\ A_r\ \sigma_r) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}_o]\!]\gamma'\delta'$$

where $\gamma' = (\gamma, c_X \mapsto (\sigma_l, \sigma_r))$ and $\delta' = (\delta, X \mapsto (A_l, A_r, R))$ which reduces in o steps to showing

$$(w', (\texttt{force}\ V_l)\ A_l\ \sigma_l, [\![\langle BC^{\sqsubseteq}_o\rangle\!\downarrow]\!][(\texttt{force}\ V_r)\ A_r\ \sigma][\gamma'_r]) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}_o]\!]\gamma'\delta'$$

by noting that by definition, $\gamma'_r = \gamma_r, c_X \mapsto \sigma_r$.

Then, we invoke monadic bind using $(w', (\texttt{force}\ V_l)\ A_l\ \sigma_l, (\texttt{force}\ V_r)\ A_r\ \sigma_r) \in \mathcal{E}^{\sim}[\![BC^{\sqsubseteq}_o]\!]\gamma'\delta'$. Let $w'' \sqsupseteq w'$ and $(w'', V_{lo}, V_{ro}) \in \mathcal{V}^{\sim}[\![BC^{\sqsubseteq}_o]\!]\gamma'\delta'$. We then need to show

$$(w'', \texttt{ret}\ V_{lo}, [\![\langle BC^{\sqsubseteq}_o\rangle\!\downarrow]\!][\texttt{ret}\ V_{ro}][\gamma'_r] \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}_o]\!]\gamma'\delta')$$

which follows by inductive hypothesis.

6. If $BC^{\sqsubseteq} = \exists^{\nu} X.BC^{\sqsubseteq}_o$, then by precision inversion also $AC^{\sqsubseteq} = \exists^{\nu} X.AC^{\sqsubseteq}_o$ and $AB^{\sqsubseteq} = \exists^{\nu} X.AB^{\sqsubseteq}_o$. We consider the upcast case, the downcast case follows by an entirely analogous argument. Given $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![\exists^{\nu} X.BC^{\sqsubseteq}_o]\!]\gamma\delta$, we need to show

$$(w, \texttt{ret}\ V_l, [\![\langle \exists^{\nu} X.BC^{\sqsubseteq}_o\rangle\!\updownarrow]\!][\texttt{ret}\ V_r][\gamma_r]) \in \mathcal{E}^{\sim}[\![\exists^{\nu} AC^{\sqsubseteq}_o]\!]\gamma\delta$$

Expanding definitions and applying anti-reduction, this reduces to showing

$$(w, \texttt{ret}\ V_l, \texttt{unpack}\ (X, y) = \texttt{ret}\ V_r; \\ \texttt{ret}\ \texttt{pack}(X, ())\texttt{thunk}\ (\lambda c_X : \text{Case}\ X.[\![\langle BC_o^{\sqsubseteq}\rangle\!\updownarrow\!]\!][(\texttt{force}\ y)\, c_X][\gamma_r])))) \\ \in \mathcal{E}^{\sim}[\![\exists^{\nu}.AC_o^{\sqsubseteq}]\!]\gamma\delta$$

By definition of $\mathcal{V}^{\sim}[\![\exists^{\nu} X.BC_o^{\sqsubseteq}]\!]\gamma\delta$, we know

$$V_l = \texttt{pack}(A_l, V_l')$$

$$V_r = \texttt{pack}(A_r, V_r')$$

and there is an associated relation $R \in \text{Rel}_{\omega}[A_l, A_r]$. Then the goal reduces to showing

$$(\texttt{pack}(A_l, V_l'), \texttt{pack}(A_l, ()\texttt{thunk}\ (\lambda c_X.[\![\langle BC_o^{\sqsubseteq}\rangle\!\updownarrow\!]\!][(\texttt{force}\ V_r')\, c_X][\gamma_r])))) \in \mathcal{V}^{\sim}[\![\exists^{\nu} X.AC_o^{\sqsubseteq}]\!]\gamma\delta$$

we choose R as the relation for X, and then we need to show (after applying anti-reduction) that for any $w' \sqsupseteq w$, $w' \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$ that

$$(w', (\texttt{force}\ V_l')\, \sigma, [\![\langle BC_o^{\sqsubseteq}\rangle\!\updownarrow\!]\!][(\texttt{force}\ V_r')\, \sigma][\gamma_r']) \in \mathcal{E}^{\sim}[\![AC_o^{\sqsubseteq}]\!]\gamma'\delta'$$

where $\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)$. and $\delta' = \delta, X \mapsto (A_l, A_r, R)$. We use the relatedness assumption and monadic bind again. Then we are given $w'' \sqsupseteq w'$, and $(V_{lo}, V_{ro}) \in \mathcal{V}^{\sim}[\![AB_o^{\sqsubseteq}]\!]\gamma'\delta'$ and need to show

$$(w'', \texttt{ret}\ V_{lo}, [\![\langle BC_o^{\sqsubseteq}\rangle\!\updownarrow\!]\!][\texttt{ret}\ V_{ro}]) \in \mathcal{E}^{\sim}[\![AC_o^{\sqsubseteq}]\!]\gamma'\delta'$$

which follows by inductive hypothesis.

□

Lemma 244 (Cast Left). *For any* $\Gamma^{\sqsubseteq}$, $\Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C$, $\Gamma \vdash AB^{\sqsubseteq} : A \sqsubseteq B$, $\Gamma^{\sqsubseteq} \vdash BC^{\sqsubseteq} : B \sqsubseteq C$ *and* $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma^{\sqsubseteq}]\!]$,

1. *If* $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![BC^{\sqsubseteq}]\!]\gamma\delta$, *then* $(w, [\![\langle AB^{\sqsubseteq}\rangle\!\downarrow\!]\!][\texttt{ret}\ V_l][\gamma_l], \texttt{ret}\ V_r) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$
2. *If* $(V_l, V_r) \in \mathcal{V}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$, *then* $(w, [\![\langle AB^{\sqsubseteq}\rangle\!\updownarrow\!]\!][\texttt{ret}\ V_l][\gamma_l], \texttt{ret}\ V_r) \in \mathcal{E}^{\sim}[\![BC^{\sqsubseteq}]\!]\gamma\delta$

Proof. By nested induction on $AB^{\sqsubseteq}$ and $AC^{\sqsubseteq}$, i.e., if $AB^{\sqsubseteq}$ becomes smaller $AC^{\sqsubseteq}$ can be anything but if $AC^{\sqsubseteq}$ becomes smaller, then $AB^{\sqsubseteq}$ must stay the same.

1. If $AC^{\sqsubseteq} \in \{\texttt{Bool}, AC_0^{\sqsubseteq} \times AC_1^{\sqsubseteq}, AC_i^{\sqsubseteq} \to AC_o^{\sqsubseteq}, \forall^{\nu} X.AC_o^{\sqsubseteq}, \exists^{\nu} X.AC_o^{\sqsubseteq}\}$, then $AB^{\sqsubseteq}$ has the same top-level connective, and the proof is symmetric to the case of Lemma 243, which always makes $AB^{\sqsubseteq}$ and $AC^{\sqsubseteq}$ smaller in uses of the inductive hypothesis.

2. If $AC^{\sqsubseteq} = ?$, then also $AB^{\sqsubseteq} = BC^{\sqsubseteq} = ?$ and the cast is trivial.
3. If $AC^{\sqsubseteq} = \mathrm{tag}_G(AG^{\sqsubseteq})$, there are two cases: either $BC^{\sqsubseteq} = ?$ or $BC^{\sqsubseteq} = \mathrm{tag}_G(BG^{\sqsubseteq})$.
   a) If $BC^{\sqsubseteq} = ?$, then $AB^{\sqsubseteq} = \mathrm{tag}_G(AG^{\sqsubseteq}) = AC^{\sqsubseteq}$. Define $\sigma_l = \gamma_l(\mathrm{case}(G))$, $\sigma_r = \gamma_r(\mathrm{case}(G))$.
      i. In the upcast case, we know $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$. In which case, $V_r = \mathtt{inj}_{\sigma_r}\ V_r'$
         A. In the $\prec$ case, we know $(w, V_l, V_r') \in \mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta$, and we need to show

$$(w, (x \leftarrow [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l]; \mathtt{ret}\ \mathtt{inj}_{\sigma_l}\ x), \mathtt{ret}\ \mathtt{inj}_{\sigma_r}\ V_r') \in \mathcal{E}^{\prec}[\![?]\!]\gamma\delta$$

            which by forward reduction is equivalent to showing

$$(w, (x \leftarrow [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l]; \mathtt{ret}\ \mathtt{inj}_{\sigma_l}\ x), x \leftarrow \mathtt{ret}\ V_r'; \mathtt{ret}\ \mathtt{inj}_{\sigma_r}\ x) \in \mathcal{E}^{\prec}[\![?]\!]\gamma\delta$$

            By inductive hypothesis, we know

$$(w, [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l], \mathtt{ret}\ V_r') \in \delta_R(\mathrm{case}(G))$$

            so can we apply monadic bind. Let $w' \sqsupseteq w$, and $(w', V_l', V_r'') \in \mathcal{V}^{\prec}[\![G]\!]\gamma\delta$. Then we need to show (after applying anti-reduction)

$$(w', \mathtt{inj}_{\sigma_l}\ V_l', \mathtt{inj}_{\sigma_r}\ V_r'') \in \mathcal{V}^{\prec}[\![?]\!]\gamma\delta$$

            To do this, we need to give a relation R such that $w' \vDash (\sigma_l, \sigma_r, R)$ and $(w', V_l', V_r'') \in \triangleright R$. Since $\gamma(\mathrm{case}(G)) = (\sigma_l, \sigma_r)$, we know $R = \lfloor \delta_R(\mathrm{case}(G)) \rfloor_{w'.j}$. And we need to show that for any $w'' \sqsupseteq w'$, that $(w'', V_l', V_r'') \in \lfloor R \rfloor_{w'.j}$. Which follows by monotonicity because $w'' \sqsupseteq w$.
         B. The $\succ$ case is slightly more complicated. This time we only know we know $(w, V_l, V_r') \in \triangleright\mathcal{V}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$ (note the $\triangleright$), and we need to show

$$(w, (x \leftarrow [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l]; \mathtt{ret}\ \mathtt{inj}_{\sigma_l}\ x), \mathtt{ret}\ \mathtt{inj}_{\sigma_r}\ V_r') \in \mathcal{E}^{\succ}[\![?]\!]\gamma\delta$$

            By Lemma 242, we know $[\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l]$ either runs to error or terminates. If it runs to an error then our goal holds. Otherwise, let $[\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l] \mapsto^* \mathtt{ret}\ V_l'$. Applying anti-reduction, we need to show

$$(w, \mathtt{inj}_{\sigma_l}\ V_l', \mathtt{inj}_{\sigma_r}\ V_r') \in \mathcal{V}^{\succ}[\![?]\!]\gamma\delta$$

            by the same reasoning as above, we need to show

$$(w, V_l', V_r') \in \triangleright\lfloor \delta_R(\mathrm{case}(G)) \rfloor_{w.j}$$

Let $w' \sqsupset w$. We need to show

$$(w', V_l', V_r') \in \lfloor \delta_R(\mathsf{case}(G)) \rfloor_{w.j}$$

By our assumption, we know

$$(w', V_l, V_r') \in \mathcal{V}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

so by inductive hypothesis, we know

$$(w', [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l], \mathtt{ret}\ V_r') \in \mathcal{E}^{\succ}[\![G]\!]\gamma\delta = \delta_R(\mathsf{case}(G))$$

And since we know $w'.\Sigma_l, [\![\langle AG^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l] \mapsto^* w'.\Sigma_l, \mathtt{ret}\ V_l'$ by Lemma 242, this means

$$(w', V_l', V_r') \in \mathcal{V}^{\succ}[\![G]\!]\gamma\delta = \delta_R(\mathsf{case}(G))$$

so the result follows by Lemma 239.

ii. For the downcast case, we know $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![?]\!]\gamma\delta$ which means there exists σ_l, σ_r, R with $w.\eta \vDash (\sigma_l, \sigma_r, R)$ and $V_l = \mathtt{inj}_{\sigma_l}\ V_l'$ and $V_r = \mathtt{inj}_{\sigma_r}\ V_r'$ and

$$(w, V_l', V_r') \in \triangleright R$$

Expanding the definition of the cast and applying anti-reduction, we need to show

$$(w, \mathtt{match}\ V_l\ \mathtt{with}\ \sigma_l\{\mathtt{inj}\ x.[\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ x] \mid \mho\}, \mathtt{ret}\ V_r) \in \mathcal{E}^{\sim}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

If $\gamma_l(\mathsf{case}(G)) = \sigma_l$, the left side errors and the result holds. Otherwise,

$$\mathtt{match}\ V_l\ \mathtt{with}\ \sigma_l\{\mathtt{inj}\ x.[\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!][\mathtt{ret}\ x] \mid \mho\} \mapsto^1 [\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!]\mathtt{ret}\ V_l'$$

A. In the $\succ$ case, we need to show

$$(w, [\![\langle AG^{\sqsubseteq}\rangle\!\downarrow]\!]\mathtt{ret}\ V_l', \mathtt{ret}\ V_r) \in \mathcal{E}^{\succ}[\![\mathrm{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

Which follows by inductive hypothesis if we can show

$$(w, V_l', V_r) \in \mathcal{V}^{\succ}[\![\mathrm{tag}_G(G)]\!]\gamma\delta$$

which reduces to showing

$$(w, V_l', V_r') \in \triangleright\mathcal{V}^{\succ}[\![G]\!]\gamma\delta$$

So let $w' \sqsupset w$. We need to show

$$(w', V_l', V_r') \in \mathcal{V}^{\succ}[\![G]\!]\gamma\delta$$

But we know $(w', V_l', V_r') \in R$ where $R = \lfloor \delta_R(\mathsf{case}(G)) \rfloor_{w.j}$ so the result follows by Lemma 240.

B. In the $\prec$ case, we check $w.j$

- If $w.j = 0$, then the left side takes 1 step so the result holds.
- Otherwise, define $w' = (w.j - 1, w.\Sigma_l, w.\Sigma_r, \lfloor w.\eta \rfloor_{w.j-1})$. Then it is sufficient to show

$$(w', [\![\langle AG^{\sqsubseteq} \rangle\!\downarrow]\!]\texttt{ret } V_l', \texttt{ret } V_r) \in \mathcal{E}^{\prec}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

To apply the inductive hypothesis we need to show

$$(w', V_l', \texttt{ret } V_r) \in \mathcal{V}^{\prec}[\![\text{tag}_G(G)]\!]\gamma\delta$$

Unrolling definitions, it is sufficient to show

$$(w', V_l', V_r') \in \mathcal{V}^{\prec}[\![G]\!]\gamma\delta$$

But we know already that $(w', V_l', V_r') \in R$ where $R = \lfloor \delta_R(\text{case}(G)) \rfloor_{w.j}$ so the result follows.

b) Finally, if $BC^{\sqsubseteq} = \text{tag}_G(BG^{\sqsubseteq})$, then $AC^{\sqsubseteq} = \text{tag}_G(AG^{\sqsubseteq})$. We consider the downcast case, the upcast case is entirely symmetric. Let $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![\text{tag}_G(BG^{\sqsubseteq})]\!]\gamma\delta$. Then we know $V_r = \texttt{inj}_{\sigma_r}\ V_r'$ where $\sigma_r = \gamma_r(\text{case}(G))$.

i. If $\sim = \prec$, we furthermore know $(w, V_l, V_r') \in \mathcal{V}^{\prec}[\![BG^{\sqsubseteq}]\!]\gamma\delta$. We need to show that

$$(w, [\![\langle AB^{\sqsubseteq} \rangle\!\downarrow]\!][\texttt{ret } V_l][\gamma_l], \texttt{ret inj}_{\sigma_r}\ V_r') \in \mathcal{E}^{\prec}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

by forward reduction it is sufficient to show

$$(w, [\![\langle AB^{\sqsubseteq} \rangle\!\downarrow]\!][\texttt{ret } V_l][\gamma_l], x \leftarrow \texttt{ret } V_r'; \texttt{ret inj}_{\sigma_r}\ x) \in \mathcal{E}^{\prec}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

We know by inductive hypothesis that

$$(w, [\![\langle AB^{\sqsubseteq} \rangle\!\downarrow]\!][\texttt{ret } V_l][\gamma_l], \texttt{ret } V_r)$$

so we can apply monadic bind. Let $w' \sqsupseteq w$, and $(w', V_l', V_r'') \in \mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta$. Then we need to show (after applying anti-reduction) that

$$(w', V_l', \texttt{inj}_{\sigma_r}\ V_r'') \in \mathcal{V}^{\prec}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

Which, unrolling the definition, is

$$(w', V_l', V_r'') \in \mathcal{V}^{\prec}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

which was our assumption.

ii. If $\sim = \succ$, we only know $(w, V_l, V_r') \in \triangleright \mathcal{V}^{\succ}[\![BG^{\sqsubseteq}]\!]\gamma\delta$ (note the $\triangleright$).

$$(w, [\![\langle AB^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_l][\gamma_l], \texttt{ret}\ \texttt{inj}_{\sigma_r}\ V_r') \in \mathcal{E}^{\succ}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

By Lemma 242, the left hand side either errors or terminates with a value.

A. If $w.\Sigma_l, [\![\langle AB^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_l][\gamma_l] \mapsto^* w.\Sigma_l, \mho$, then the result holds.

B. If $w.\Sigma_l, [\![\langle AB^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_l][\gamma_l] \mapsto^* w.\Sigma_l, \texttt{ret}\ V_l'$, then we need to show that

$$(w, V_l', \texttt{inj}_{\sigma_r}\ V_r') \in \mathcal{V}^{\succ}[\![\text{tag}_G(AG^{\sqsubseteq})]\!]\gamma\delta$$

Which unrolls to

$$(w, V_l', V_r') \in \triangleright \mathcal{V}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

So let $w' \sqsupset w$. We need to show

$$(w', V_l', V_r') \in \mathcal{V}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

By inductive hypothesis, we know

$$(w', [\![\langle AB^{\sqsubseteq}\rangle\downarrow]\!][\texttt{ret}\ V_l][\gamma_l], \texttt{ret}\ V_r') \in \mathcal{E}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

and so by determinism of evaluation, we know

$$(w', V_l', V_r') \in \mathcal{V}^{\succ}[\![AG^{\sqsubseteq}]\!]\gamma\delta$$

so the result holds.

□

Lemma 245.

$$\Gamma_1^{\sqsubseteq}, x : A^{\sqsubseteq}, \Gamma_2^{\sqsubseteq} \vDash x \sqsubseteq_{\sim} x \in A^{\sqsubseteq}$$

Proof. We need to show

$$(w, \texttt{ret}\ V_l, \texttt{ret}\ V_r) \in \mathcal{E}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$$

where $V_i = \gamma_i(x)$. Since both sides are values, it is sufficient to show

$$(w, \gamma_l(x), \gamma_r(x)) \in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$$

By definition of $\mathcal{G}^{\sim}[\![\Gamma_1^{\sqsubseteq}, x : A^{\sqsubseteq}, \Gamma_2^{\sqsubseteq}]\!]$, we know $\gamma = \gamma_1, x \mapsto (V_l, V_r), \gamma_2$ and $\delta = \delta_1, \delta_2$ where $(w, \gamma_1, \delta_2) \in \mathcal{G}^{\sim}[\![\Gamma_1^{\sqsubseteq}]\!]$ and $(w, V_l, V_r) \in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma_1\delta_1$.

Then the result follows because $\mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma_1\delta_1 = \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$ by Lemma 241 □

Lemma 246 (Compatibility: Upcast Left)**.**

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in AC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma_l \vDash AB^{\sqsubseteq} : A \sqsubseteq B \qquad \Gamma^{\sqsubseteq} \vDash BC^{\sqsubseteq} : B \sqsubseteq C}{\Gamma^{\sqsubseteq} \vDash \langle AB^{\sqsubseteq}\rangle\!\updownarrow M_l \sqsubseteq M_r \in BC^{\sqsubseteq}}$$

Proof. We need to show

$$(w, [\![\langle AB^{\sqsubseteq}\rangle\!\updownarrow]\!][[\![M_l]\!]][\gamma_l][\delta_l], [\![M_r]\!][\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![BC^{\sqsubseteq}]\!]\gamma\delta$$

By assumption, we know

$$(w, [\![M_l]\!][\gamma_l][\delta_l], [\![M_r]\!][\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$$

So we apply monadic bind. Let $w' \sqsupseteq w$ and $(w', V_l, V_r) \in \mathcal{V}^{\sim}[\![AC^{\sqsubseteq}]\!]\gamma\delta$. We need to show

$$(w', [\![\langle AB^{\sqsubseteq}\rangle\!\updownarrow]\!][\mathtt{ret}\ V_l][\gamma_l], \mathtt{ret}\ V_r) \in \mathcal{E}^{\sim}[\![BC^{\sqsubseteq}]\!]\gamma\delta$$

which follows by Lemma 244. □

Lemma 247 (Compatibility: Downcast Left)**.**

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in BC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma_l \vdash AB^{\sqsubseteq} : A \sqsubseteq B \Gamma^{\sqsubseteq} \vdash BC^{\sqsubseteq} : B \sqsubseteq C}{\Gamma^{\sqsubseteq} \vDash \langle AB^{\sqsubseteq}\rangle\!\downarrow M_l \sqsubseteq M_r \in AC^{\sqsubseteq}}$$

Proof. By same argument as Lemma 246. □

Lemma 248 (Compatibility: Upcast Right)**.**

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in AB^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma_r \vdash BC^{\sqsubseteq} : B \sqsubseteq C \qquad \Gamma^{\sqsubseteq} \vdash AB^{\sqsubseteq} : A \sqsubseteq B}{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq \langle BC^{\sqsubseteq}\rangle\!\updownarrow M_r \in AC^{\sqsubseteq}}$$

Proof. By same argument as Lemma 246, but using Lemma 243. □

Lemma 249.

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in AC^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash AC^{\sqsubseteq} : A \sqsubseteq C \qquad \Gamma^{\sqsubseteq} : \Gamma_l \sqsubseteq \Gamma_r \qquad \Gamma_r \vdash BC^{\sqsubseteq} : B \sqsubseteq C \qquad \Gamma^{\sqsubseteq} \vdash AB^{\sqsubseteq} : A \sqsubseteq B}{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq \langle BC^{\sqsubseteq}\rangle\!\downarrow M_r \in AB^{\sqsubseteq}}$$

Proof. By same argument as Lemma 246, but using Lemma 243. □

Lemma 250. *If* $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_1^{\sqsubseteq}, X \cong A^{\sqsubseteq}, \Gamma_2^{\sqsubseteq}]\!]$ *of* $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_1^{\sqsubseteq}, X, \Gamma_2^{\sqsubseteq}]\!]$*, then*

$$\mathcal{V}^{\sim}[\![X]\!]\gamma\delta = \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma\delta$$

Proof. By definition, $\gamma = \gamma_1, c_X \mapsto (\sigma_l, \sigma_r), \gamma_2$ and $\delta = \delta_1, X \mapsto (A_l, A_r, R), \delta_2$, where $(w, \gamma_1, \delta_1) \in \mathcal{G}^\sim[\![\Gamma^{\sqsubseteq}_1]\!]$. Then

$$\mathcal{V}^\sim[\![X]\!]\gamma\delta = \delta(X) = \mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma_1\delta_1$$

So it is sufficient to show

$$\mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma_1\delta_1 = \mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta$$

which follows by Lemma 241. □

Lemma 251. *Seal*

$$\frac{(X \cong A^{\sqsubseteq}) \in \Gamma^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash \mathtt{seal}_X M_l \sqsubseteq \mathtt{seal}_X M_r \in X}$$

Proof. Assume $(M_l[\gamma_l][\delta_l], M_r[\gamma_l][\delta_l]) \in \mathcal{E}^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta$. We need to show

$$(M_l[\gamma_l][\delta_l], M_r[\gamma_l][\delta_l]) \in \mathcal{E}^\sim[\![X]\!]\gamma\delta$$

This follows immediately from Lemma 250 □

Lemma 252.

$$\frac{(X \cong A^{\sqsubseteq}) \in \Gamma^{\sqsubseteq}, \Gamma^{\sqsubseteq\prime} \qquad \Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in X}{\Gamma^{\sqsubseteq} \vDash \mathtt{unseal}_X M_l \sqsubseteq \mathtt{unseal}_X M_r \in A^{\sqsubseteq}}$$

Proof. By same reasoning as the seal case. □

Lemma 253.

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in ? \qquad \Gamma^{\sqsubseteq} \vdash G}{\Gamma^{\sqsubseteq} \vDash \mathtt{is(G)?}\ M_l \sqsubseteq \mathtt{is(G)?}\ M_r \in \mathtt{Bool}}$$

Proof. Define $M'_l = [\![M_l]\!][\gamma_l][\delta_l]$ and similarly for M_r. Then we need to show

$$\begin{aligned}&(w,\\ &r \leftarrow M'_l; \mathtt{match}\ r\ \mathtt{with}\ \mathrm{case}(G)\{\mathtt{inj}\ y.\mathtt{ret}\ \mathtt{true} \mid \mathtt{ret}\ \mathtt{false}\},\\ &r \leftarrow M'_r; \mathtt{match}\ r\ \mathtt{with}\ \mathrm{case}(G)\{\mathtt{inj}\ y.\mathtt{ret}\ \mathtt{true} \mid \mathtt{ret}\ \mathtt{false}\})\\ &\in \mathcal{E}^\sim[\![\mathtt{Bool}]\!]\gamma\delta\end{aligned}$$

Since we know by assumption that $(w, M'_l, M'_r) \in \mathcal{E}^\sim[\![?]\!]\gamma\delta$, we can apply monadic bind. Suppose $w' \sqsupseteq w$ and $(w', V_l, V_r) \in \mathcal{V}^\sim[\![?]\!]\gamma\delta$. Then we need to show (after applying anti-reduction) that

$$\begin{aligned}&(w',\\ &\mathtt{match}\ V_l\ \mathtt{with}\ \mathrm{case}(G)\{\mathtt{inj}\ y.\mathtt{ret}\ \mathtt{true} \mid \mathtt{ret}\ \mathtt{false}\},\\ &\mathtt{match}\ V_r\ \mathtt{with}\ \mathrm{case}(G)\{\mathtt{inj}\ y.\mathtt{ret}\ \mathtt{true} \mid \mathtt{ret}\ \mathtt{false}\})\\ &\in \mathcal{E}^\sim[\![\mathtt{Bool}]\!]\gamma\delta\end{aligned}$$

By definition of $\mathcal{V}^{\sim}[\![?]\!]\gamma\delta$, we know for some $w'.\eta \vDash (\sigma_l, \sigma_r, R)$ that $V_l = \mathtt{inj}_{\sigma_l}\ V_l'$, $V_r = \mathtt{inj}_{\sigma_l}\ V_r'$ and $(w', V_l', V_r') \in \triangleright R$.

By definition of case(G), if $\sigma_l = \text{case}(G)$, then so does σ_r and vice-versa. If this is the case, both sides reduce to `ret true` and otherwise both sides reduce to `ret false`. Then the result holds by anti-reduction. □

Lemma 254.

$$\Gamma^{\sqsubseteq} \vDash \mathit{true} \sqsubseteq \mathit{true} \in \mathit{Bool} \qquad \Gamma^{\sqsubseteq} \vDash \mathit{false} \sqsubseteq \mathit{false} \in \mathit{Bool}$$

Proof. The result $(w, \mathtt{ret\ true}, \mathtt{ret\ true}) \in \mathcal{E}^{\sim}[\![\mathtt{Bool}]\!]\gamma\delta$ follows because $(w, \mathtt{true}, \mathtt{true}) \in \mathcal{V}^{\sim}[\![\mathtt{Bool}]\!]\gamma\delta$. Similarly for `false`. □

Lemma 255.

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in \mathit{Bool} \quad \Gamma^{\sqsubseteq} \vDash N_{lt} \sqsubseteq N_{rt} \in B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vDash N_{lf} \sqsubseteq N_{rf} \in B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash \mathit{if}\ M_l\ \mathit{then}\ N_{lt}\ \mathit{else}\ N_{lf} \sqsubseteq \mathit{if}\ M_r\ \mathit{then}\ N_{rt}\ \mathit{else}\ N_{rf} \in B^{\sqsubseteq}}$$

Proof. Define $M_l' = [\![M_l]\!][\gamma_l][\delta_l]$ and similarly for the rest of the subterms. Then we need to show

$$\begin{aligned}&(w,\\ &x \leftarrow M_l'; \mathtt{case}\ x\{x_t.N_{lt}' \mid x_f.N_{lf}'\},\\ &x \leftarrow M_r'; \mathtt{case}\ x\{x_t.N_{rt}' \mid x_f.N_{rf}'\})\\ &\in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta\end{aligned}$$

By assumption and weakening, $(w, M_l', M_r') \in \mathcal{E}^{\sim}[\![\mathtt{Bool}]\!]\gamma\delta$, and we apply monadic bind. Suppose $w' \sqsupseteq w$ and $(w', V_l, V_r) \in \mathcal{V}^{\sim}[\![\mathtt{Bool}]\!]\gamma\delta$. We need to show

$$\begin{aligned}&(w,\\ &x \leftarrow \mathtt{ret}\ V_l; \mathtt{case}\ x\{x_t.N_{lt}' \mid x_f.N_{lf}'\},\\ &x \leftarrow \mathtt{ret}\ V_r; \mathtt{case}\ x\{x_t.N_{rt}' \mid x_f.N_{rf}'\})\\ &\in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta\end{aligned}$$

By definition either $V_l = V_r = \mathtt{true}$ or $V_l = V_r = \mathtt{false}$. Without loss of generality assume it is `true`. Then

$$x \leftarrow \mathtt{ret}\ V_l; \mathtt{case}\ x\{x_t.N_{lt}' \mid x_f.N_{lf}'\} \mapsto^0 N_{lt}'$$

and similarly for the right side. By anti-reduction (Lemma 234), it is sufficient to show

$$(w', N_{lt}', N_{rt}') \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta$$

which follows by hypothesis. □

Lemma 256. *Product intro*

$$\frac{\Gamma^{\sqsubseteq} \vDash M_{l0} \sqsubseteq M_{r0} \in A_0^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vDash M_{l1} \sqsubseteq M_{r1} \in A_1^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash (M_{l0}, M_{l1}) \sqsubseteq (M_{r0}, M_{r1}) \in A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}}$$

Proof. We need to show that

$$\begin{array}{l}(w, x \leftarrow [\![M_{l0}]\!][\gamma_l][\delta_l];, x \leftarrow [\![M_{r0}]\!][\gamma_r][\delta_r];) \in \mathcal{E}^{\sim}[\![A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}]\!]\gamma\delta \\ \quad y \leftarrow [\![M_{l1}]\!][\gamma_l][\delta_l];\ y \leftarrow [\![M_{r1}]\!][\gamma_r][\delta_r]; \\ \quad \texttt{ret}\ (x, y) \qquad\qquad \texttt{ret}\ (x, y)\end{array}$$

By inductive hypothesis and weakening, we know $(w, [\![M_{l0}]\!][\gamma_l][\delta_l], [\![M_{r0}]\!][\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![A_0^{\sqsubseteq}]\!]\gamma\delta$. Applying monadic bind we get some $w' \sqsupseteq w$ and $(w', V_{l0}, V_{r0}) \in \mathcal{V}^{\sim}[\![A_0^{\sqsubseteq}]\!]\gamma\delta$ and applying anti-reduction, we need to show

$$\begin{array}{l}(w', y \leftarrow [\![M_{l1}]\!][\gamma_l][\delta_l];, y \leftarrow [\![M_{r1}]\!][\gamma_r][\delta_r];) \in \mathcal{E}^{\sim}[\![A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}]\!]\gamma\delta \\ \quad \texttt{ret}\ (V_{l0}, y) \qquad\qquad \texttt{ret}\ (V_{r0}, y)\end{array}$$

By inductive hypothesis, weakening and monotonicity, we know $(w', [\![M_{l1}]\!][\gamma_l][\delta_l], [\![M_{r1}]\!][\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![A_1^{\sqsubseteq}]\!]\gamma\delta$. Applying monadic bind we get some $w'' \sqsupseteq w'$ and $(w'', V_{l1}, V_{r1}) \in \mathcal{V}^{\sim}[\![A_1^{\sqsubseteq}]\!]\gamma\delta$ and applying anti-reduction we need to show

$$(w'', (V_{l0}, V_{l1}), (V_{r0}, V_{r1})) \in \mathcal{V}^{\sim}[\![A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}]\!]\gamma\delta$$

That is that for each $i \in \{0, 1\}$ that

$$(w'', V_{li}, V_{ri}) \in \mathcal{V}^{\sim}[\![A_i^{\sqsubseteq}]\!]\gamma\delta$$

which follows by assumption and monotonicity. □

Lemma 257. *Product elim*

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A_0^{\sqsubseteq} \times A_1^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, x : A_0^{\sqsubseteq}, y : A_1^{\sqsubseteq} \vDash N_l \sqsubseteq N_r \in B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash \texttt{let}\ (x, y) = M_l; N_l \sqsubseteq \texttt{let}\ (x, y) = M_r; N_r \in B^{\sqsubseteq}}$$

Proof. We need to show

$$\begin{array}{l}(w, z \leftarrow [\![M_l]\!][\gamma_l][\delta_l];, z \leftarrow [\![M_r]\!][\gamma_r][\delta_r];) \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta \\ \quad \texttt{let}\ (x, y) = z; \quad \texttt{let}\ (x, y) = z; \\ \quad [\![N_l]\!][\gamma_l][\delta_l] \qquad\qquad [\![N_r]\!][\gamma_r][\delta_r]\end{array}$$

First, by inductive hypothesis and weakening, we know

$$(w, [\![M_l]\!][\gamma_l][\delta_l], [\![M_r]\!][\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![A_0^{\sqsubseteq} \times A_1^{\sqsubseteq}]\!]\gamma\delta$$

Applying monadic bind, we get some $w' \sqsupseteq w$ with $(w', V_{l0}, V_{r0}) \in \mathcal{V}^{\sim}[\![A_0^{\sqsubseteq}]\!]\gamma\delta$ and $(w', V_{l1}, V_{r1}) \in \mathcal{V}^{\sim}[\![A_1^{\sqsubseteq}]\!]\gamma\delta$, and applying anti-reduction we need to show

$$(w', [\![N_l]\!][\gamma_l'][\delta_l], [\![N_r]\!][\gamma_r'][\delta_r]) \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta$$

Where we define $\gamma' = \gamma, x \mapsto (V_{l0}, V_{r0}), y \mapsto (V_{l1}, V_{r1})$. By weakening, it is sufficient to show

$$(w', [\![N_l]\!][\gamma_l'][\delta_l], [\![N_r]\!][\gamma_r'][\delta_r]) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma'\delta$$

which follows by inductive hypothesis if we can show

$$(w', \gamma', \delta) \in \mathcal{G}^\sim[\![\Gamma_p, \Gamma^\sqsubseteq, \Gamma^\sqsubseteq_M, \Gamma^\sqsubseteq_N, x : A_0^\sqsubseteq, y : A_1^\sqsubseteq]\!]$$

which follows by definition and monotonicity. □

Lemma 258. *Fun intro*

$$\frac{\Gamma^\sqsubseteq, x : A^\sqsubseteq \vDash M_l \sqsubseteq M_r \in B^\sqsubseteq}{\Gamma^\sqsubseteq \vDash \lambda x : A_l.M_l \sqsubseteq \lambda x : A_l.M_l \in A^\sqsubseteq \to B^\sqsubseteq}$$

Proof. It is sufficient to show

$$(w, \texttt{thunk}\ (\lambda x : A_l[\delta_l].M_l'), \texttt{thunk}\ (\lambda x : A_r[\delta_r].M_r')) \in \mathcal{V}^\sim[\![A^\sqsubseteq \to B^\sqsubseteq]\!]\gamma\delta$$

where $M_l' = [\![M_l]\!][\gamma_l][\delta_l]$ and similarly define M_r'. Suppose $w' \sqsupseteq w$ and $(w', V_l, V_r) \in \mathcal{V}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$. We need to show

$$\begin{aligned}&(w',\\ &\texttt{force}\ (\texttt{thunk}\ (\lambda x : A_l[\delta_l].[\![M_l]\!][\gamma_l][\delta_l]))\ V_l,\\ &\texttt{force}\ (\texttt{thunk}\ (\lambda x : A_r[\delta_r].[\![M_r]\!][\gamma_r][\delta_r]))\ V_r)\\ &\in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta\end{aligned}$$

By anti-reduction it is sufficient to show

$$(w', [\![M_l]\!][\gamma_l'][\delta_l], [\![M_r]\!][\gamma_r'][\delta_r]) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta$$

where $\gamma' = \gamma, x \mapsto (V_l, V_r)$. By monotonicity, we have $(w', \gamma', \delta) \in \mathcal{G}^\sim[\![\Gamma_p, \Gamma^\sqsubseteq]\!]$ so the result follows by hypothesis. □

Lemma 259.

$$\frac{\Gamma^\sqsubseteq \vdash M_l \sqsubseteq M_r : A^\sqsubseteq \to B^\sqsubseteq \qquad \Gamma^\sqsubseteq \vdash N_l \sqsubseteq N_r : A^\sqsubseteq}{\Gamma^\sqsubseteq \vdash M_l\ N_l \sqsubseteq M_r\ N_r : B^\sqsubseteq}$$

Proof. Define $M_l' = [\![M_l]\!][\gamma_l][\delta_l]$, etc. We need to show

$$(w, f \leftarrow M_l'; x \leftarrow N_l'; \texttt{force}\ f\ x f \leftarrow M_r'; x \leftarrow N_r'; \texttt{force}\ f\ x) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta$$

We apply monadic bind with $(w, M_l', M_r') \in \mathcal{E}^\sim[\![A^\sqsubseteq \to B^\sqsubseteq]\!]\gamma\delta$ which holds by hypothesis and weakening. Suppose $w' \sqsupseteq w$, with anti-reduction it is sufficient to show

$$(w', x \leftarrow N_l'; \texttt{force}\ V_l\ x, x \leftarrow N_r'; \texttt{force}\ V_r\ x) \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma\delta$$

where $(w', V_l, V_r) \in \mathcal{V}^\sim[\![A^\sqsubseteq \to B^\sqsubseteq]\!]\gamma\delta$. Then we apply monadic bind with $(w', N_l', N_r') \in \mathcal{E}^\sim[\![A^\sqsubseteq]\!]\gamma\delta$ which holds by hypothesis, weakening

and monotonicity. Suppose $w'' \sqsupseteq w'$; with anti-reduction it is sufficient to show

$$(w'', \texttt{force}\ V_l\ V_l', \texttt{force}\ V_r\ V_r') \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma\delta$$

Which follows by definition of the value relation and transitivity of $\sqsupseteq$. □

Lemma 260. *Forall intro*

$$\frac{\Gamma^{\sqsubseteq}, X \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash \Lambda^{\nu} X.M_l \sqsubseteq \Lambda^{\nu} X.M_r \in \forall^{\nu} X.A^{\sqsubseteq}}$$

Proof. Define $M_i' = [\![M_i]\!][\gamma_i][\delta_i]$ It is sufficient to show

$$(w, \texttt{thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.M_l'), \texttt{thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.M_r')) \in \mathcal{V}^{\sim}[\![\forall^{\nu} X.A^{\sqsubseteq}]\!]\gamma\delta$$

Given $w' \sqsupseteq w$, and $R \in \text{Rel}_n[B_l, B_r]$ and $w'.\eta \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$, we need to show

$$\begin{aligned}&(w',\\ &\texttt{force thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.M_l')\ B_l\ \sigma_l\\ &\texttt{force thunk}\ (\Lambda X.\lambda c_X : \text{Case}\ X.M_r')\ B_r\ \sigma_r)\\ &\in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma'\delta'\end{aligned}$$

where $\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)$ and $\delta' = \delta, X \mapsto (B_l, B_r, R)$. By anti-reduction it is sufficient to show

$$(w', [\![M_l]\!][\gamma_l'][\delta_l'], [\![M_r]\!][\gamma_r'][\delta_r']) \in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma'\delta'$$

which follows by hypothesis if since $(w', \gamma', \delta') \in \mathcal{G}^{\sim}[\![\Gamma_p, \Gamma^{\sqsubseteq}, X]\!]$ by definition and monotonicity. □

Lemma 261. *Forall Elim*

$$\frac{\begin{array}{c}\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in \forall^{\nu} X.A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r\\ \Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq}, x : A^{\sqsubseteq} \vDash N_l \sqsubseteq N_r \in B^{\sqsubseteq\prime}\end{array}}{\Gamma^{\sqsubseteq} \vDash \texttt{let}\ x = M_l\{X \cong B_l\}; N_l \sqsubseteq \texttt{let}\ x = M_l\{X \cong B_l\}; N_l \in B^{\sqsubseteq\prime}}$$

Proof. Define $M_i' = [\![M_i]\!][\gamma_i][\delta_i]$, $N_i' = [\![N_i]\!][\gamma_i][\delta_i]$ and $B_i = [\![B_i]\!][\delta_i]$. We need to show

$$\begin{aligned}&(w,\\ &f \leftarrow M_l'; \texttt{newcase}_{B_l'}\ c_X; x \leftarrow \texttt{force}\ f\ [B_l']\ c_X; N_l',\\ &f \leftarrow M_r'; \texttt{newcase}_{B_r'}\ c_X; x \leftarrow \texttt{force}\ f\ [B_r']\ c_X; N_r')\\ &\in \mathcal{E}^{\sim}[\![B^{\sqsubseteq\prime}]\!]\gamma\delta\end{aligned}$$

We know by hypothesis that $(w, M_l', M_r') \in \mathcal{E}^\sim[\![\forall^\nu X.A^{\sqsubseteq}]\!]\gamma\delta$, so we can apply monadic bind. Suppose $w' \sqsupseteq w$ and $(w', V_l, V_r) \in \mathcal{V}^\sim[\![\forall^\nu X.A^{\sqsubseteq}]\!]\gamma\delta$. Applying anti-reduction, we need to show

$$\begin{array}{l}(w',\\ \texttt{newcase}_{B_l'}\ c_X; x \leftarrow \texttt{force}\ V_l\ [B_l']\ c_X; N_l',\\ \texttt{newcase}_{B_r'}\ c_X; x \leftarrow \texttt{force}\ V_r\ [B_r']\ c_X; N_r')\\ \in \mathcal{E}^\sim[\![B^{\sqsubseteq\prime}]\!]\gamma\delta\end{array}$$

Define $\sigma_i = w'.\Sigma_l.size$, and $w'' = (w'.j, (w'.\Sigma_l, [\![A_l]\!][\delta]), (w'.\Sigma_r, [\![A_r]\!][\delta]), \eta \boxplus (\sigma_l, \sigma_r, \lfloor \mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma\delta \rfloor'_{w}.j))$ Then we have

$$w'.\Sigma_i \triangleright \texttt{newcase}_{B_i'}\ c_X; x \leftarrow \texttt{force}\ V_i\ [B_i']\ c_X; N_i' \mapsto^0 w''.\Sigma_i \triangleright x \leftarrow \texttt{force}\ V_i\ [B_i']\ \sigma_i; N_i'$$

So by anti-reduction, it is sufficient to show

$$(w'', x \leftarrow \texttt{force}\ V_l\ [B_l']\ \sigma_l; N_l', x \leftarrow \texttt{force}\ V_r\ [B_r']\ \sigma_r; N_r') \in \mathcal{E}^\sim[\![B^{\sqsubseteq\prime}]\!]\gamma\delta$$

Since $(w', V_l, V_r) \in \mathcal{V}^\sim[\![\forall^\nu X.A^{\sqsubseteq}]\!]\gamma\delta$, by definition we know

$$(w'', \texttt{force}\ V_l\ [B_l']\ \sigma_l, \texttt{force}\ V_r\ [B_r']\ \sigma_r) \in \mathcal{E}^\sim[\![A^{\sqsubseteq}]\!]\gamma'\delta'$$

where $\gamma' = \gamma, c_x \mapsto (\sigma_l, \sigma_r)$ and $\delta' = \delta, X \mapsto (B_l', B_r', R)$. Next, note that since X is free in $B^{\sqsubseteq\prime}$, that by weakening, $\mathcal{E}^\sim[\![B^{\sqsubseteq\prime}]\!]\gamma'\delta' = \mathcal{E}^\sim[\![B^{\sqsubseteq\prime}]\!]\gamma\delta$. Therefore we can apply monadic bind. Suppose $w''' \sqsupseteq w''$ and $(w''', V_l', V_r') \in \mathcal{V}^\sim[\![A^{\sqsubseteq}]\!]\gamma'\delta'$, then by anti-reduction, it is sufficient to show

$$(w''', N_l'[x \mapsto V_l'], N_r'[x \mapsto V_r']) \in \mathcal{E}^\sim[\![B^{\sqsubseteq\prime}]\!]\gamma'\delta'$$

which follows by assumption about N_l, N_r. □

Lemma 262.

$$\frac{\Gamma^{\sqsubseteq}, X \cong B^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vdash B^{\sqsubseteq} : B_l \sqsubseteq B_r}{\Gamma^{\sqsubseteq} \vDash \mathit{pack}^\nu(X \cong B_l, M_l) \sqsubseteq \mathit{pack}^\nu(X \cong B_r, M_r) \in \exists^\nu X.A^{\sqsubseteq}}$$

Proof. Recall the translation is

$$[\![\texttt{pack}^\nu(X \cong B_i, M_i)]\!] = \texttt{ret}\ (\texttt{pack}([\![B_i]\!], \texttt{thunk}\ \lambda c_X : \text{Case}\ [\![B_i]\!].[\![M_i]\!])\ \texttt{as}\ [\![\exists^\nu X.A_i]\!])$$

where $\Gamma^{\sqsubseteq}, X \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$. So it is sufficient to show

$$\begin{array}{l}(w,\\ \texttt{pack}([\![B_l]\!][\delta_l], \texttt{thunk}\ \lambda c_X : \text{Case}\ ([\![B_l]\!][\delta_l]).[\![M_l]\!][\gamma_l][\delta_l])\ \texttt{as}\ [\![\exists^\nu X.A^{\sqsubseteq}]\!][\delta_l],\\ \texttt{pack}([\![B_r]\!][\delta_r], \texttt{thunk}\ \lambda c_X : \text{Case}\ ([\![B_r]\!][\delta_r]).[\![M_r]\!][\gamma_r][\delta_r])\ \texttt{as}\ [\![\exists^\nu X.A^{\sqsubseteq}]\!][\delta_r])\\ \in \mathcal{V}^\sim[\![\exists^\nu X.A^{\sqsubseteq}]\!]\gamma\delta\end{array}$$

For the relation, we pick $R = \mathcal{V}^\sim[\![B^{\sqsubseteq}]\!]\gamma\delta$. Let $w' \sqsupseteq w$ and σ_l, σ_r be seals such that $w'.\eta \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$. Then we need to show

$$\begin{array}{l}(w,\\ \texttt{force}\ (\texttt{thunk}\ \lambda c_X : \text{Case}\ ([\![B_l]\!][\delta_l]).[\![M_l]\!][\gamma_l][\delta_l])\ \sigma_l,\\ \texttt{force}\ (\texttt{thunk}\ \lambda c_X : \text{Case}\ ([\![B_r]\!][\delta_r]).[\![M_r]\!][\gamma_r][\delta_r])\ \sigma_r)\\ \in \mathcal{E}^\sim[\![A^{\sqsubseteq}]\!]\gamma'\delta'\end{array}$$

where

$$\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)$$
$$\delta' = \delta, X \mapsto (\llbracket B_l \rrbracket[\delta_l], \llbracket B_l \rrbracket[\delta_l], R)$$

By anti reduction this reduces to showing

$$(w', \llbracket M_l \rrbracket[\gamma_l'][\delta_l'], \llbracket M_r \rrbracket[\gamma_r'][\delta_r']) \in \mathcal{E}^{\sim}\llbracket A^{\sqsubseteq} \rrbracket \gamma' \delta'$$

which follows by assumption. □

Lemma 263.

$$\frac{\Gamma^{\sqsubseteq} \vDash M_l \sqsubseteq M_r \in \exists^{\nu} X.A^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq}, X, x : A^{\sqsubseteq} \vDash N_l \sqsubseteq N_r \in B^{\sqsubseteq} \qquad \Gamma^{\sqsubseteq} \vDash B^{\sqsubseteq}}{\Gamma^{\sqsubseteq} \vDash \mathit{unpack}\ (X, x) = M_l; N_l \sqsubseteq \mathit{unpack}\ (X, x) = M_r; N_r \in B^{\sqsubseteq}}$$

Proof. Recall the translation:

$$\begin{aligned}\llbracket \mathtt{unpack}\ (X, x) = M_i; N_i \rrbracket = {} & p \leftarrow \llbracket M_i \rrbracket; \\ & \mathtt{unpack}\ (X, f) = p; \\ & \mathtt{newcase}_X\ c_X; \\ & x \leftarrow (\mathtt{force}\ f\, c_X); \\ & \llbracket N_i \rrbracket\end{aligned}$$

Let $(w, \gamma, \delta) \in \mathcal{G}^{\sim}\llbracket \Gamma_p, \Gamma^{\sqsubseteq}, \Gamma^{\sqsubseteq}_M, \Gamma^{\sqsubseteq}_N \rrbracket$. By assumption (and weakening), we know $(w, \llbracket M_l \rrbracket[\gamma_l][\delta_l], \llbracket M_r \rrbracket[\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}\llbracket \exists^{\nu} X.A^{\sqsubseteq} \rrbracket$. We apply monadic bind. Let $w' \sqsupseteq w$ and $(w', V_{\exists^{\nu} l}, V_{\exists^{\nu} r}) \in \mathcal{V}^{\sim}\llbracket \exists^{\nu} X.A^{\sqsubseteq} \rrbracket$.

Then $V_{\exists^{\nu} l} = \mathtt{pack}(A_{Xl}, V_{fl})$ and $V_{\exists^{\nu} r} = \mathtt{pack}(A_{Xr}, V_{fr})$ with some associated relation $R \in \mathrm{Rel}[A_{Xl}, A_{Xr}]$. Then by anti-reduction we need to show

$$\begin{array}{lll}(w', \mathtt{newcase}_{A_{Xl}}\ c_X; & , \mathtt{newcase}_{A_{Xr}}\ c_X; &) \in \mathcal{E}^{\sim}\llbracket B^{\sqsubseteq} \rrbracket \gamma \delta \\ \quad x \leftarrow (\mathtt{force}\ V_{fl}\, c_X); & x \leftarrow (\mathtt{force}\ V_{fr}\, c_X); & \\ \quad \llbracket N_l \rrbracket[\gamma_l][\delta_l] & \llbracket N_r \rrbracket[\gamma_l][\delta_l] & \end{array}$$

Define

$$w'' = (w'.j, (w'.\Sigma_l, A_{Xl}), (w'.\Sigma_r, A_{Xr}), w'.\eta \uplus (A_{Xl}, A_{Xr}, \lfloor R \rfloor_{w'.j}))$$
$$\gamma' = \gamma, c_X \mapsto (w'.\Sigma_l.\mathit{size}, w'.\Sigma_r.\mathit{size})$$
$$\delta' = \delta, X \mapsto (A_{Xl}, A_{Xr}, R)$$

Then by anti-reduction it is sufficient to show

$$(w'', (x \leftarrow (\mathtt{force}\ V_{fl}\, \sigma_l); \llbracket N_l \rrbracket[\gamma_l'][\delta_l']), (x \leftarrow (\mathtt{force}\ V_{fr}\, \sigma_r); \llbracket N_r \rrbracket[\gamma_r'][\delta_r'])) \in \mathcal{E}^{\sim}\llbracket B^{\sqsubseteq} \rrbracket \gamma' \delta'$$

By assumption (and monotonicity), $(w'', \mathtt{force}\ V_{fl}\, \sigma_l, \mathtt{force}\ V_{fr}\, \sigma_r) \in \mathcal{E}^{\sim}\llbracket A^{\sqsubseteq} \rrbracket \gamma' \delta'$, so we can apply monadic bind. Let $w''' \sqsupseteq w''$ and

$(w''', V_{Al}, V_{Ar}) \in \mathcal{V}^{\sim}[\![A^{\sqsubseteq}]\!]\gamma'\delta'$. Then after anti-reduction we need to show

$$(w''', [\![N_l]\!][\gamma_l''][\delta_l'], [\![N_r]\!][\gamma_r''][\delta_r']) \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma'\delta'$$

where $\gamma'' = \gamma', x \mapsto (V_{Al}, V_{Ar})$. By weakening this is equivalent to

$$(w''', [\![N_l]\!][\gamma_l''][\delta_l'], [\![N_r]\!][\gamma_r''][\delta_r']) \in \mathcal{E}^{\sim}[\![B^{\sqsubseteq}]\!]\gamma''\delta'$$

which follows by assumption. □

10.6.3.3 *Graduality Proof*

Lemma 264. *If* $C : (\Gamma \vdash \cdot : A) \Rightarrow (\Gamma' \vdash \cdot : B)$ *and* $\Gamma \vDash M_l \sqsubseteq M_r \in A$, *then* $\Gamma' \vDash C[M_l] \sqsubseteq C[M_r] \in B$.

Proof. By induction on C, using a corresponding compatibility lemma in each case. □

We define contextual error approximation, following [56]:

Definition 265. Let $\Gamma \vdash M_1 : A; \Gamma_M$ and $\Gamma \vdash M_2 : A; \Gamma_M$. Then we say M_1 (contextually) *error approximates* M_2, written $\Gamma \vDash M_1 \sqsubseteq^{ctx} M_2 \in A; \Gamma_M$ when for any context $C : (\Gamma \vdash A; \Gamma_M) \Rightarrow (\cdot \vdash B; \cdot)$, all of the following hold:

1. If $C[M_1] \Uparrow$ then $C[M_2] \Uparrow$
2. If $C[M_1] \Downarrow V_1$ then there exists V_2 such that $C[M_2] \Downarrow V_2$.

Definition 266. If $\Gamma \vdash M_1 : A; \Gamma_M$ and $\Gamma \vdash M_2 : A; \Gamma_M$, then M_1 and M_2 are *contextually equivalent*, $\Gamma \vDash M_1 \approx^{\text{ctx}} M_2 \in A; \Gamma_M$, when for any context $C : (\Gamma \vdash A; \Gamma_M) \Rightarrow (\cdot \vdash B; \cdot)$, both diverge $C[M_1], C[M_2] \Uparrow$, or error $C[M_1], C[M_2] \Downarrow \mho$ or terminate successfully $C[M_1] \Downarrow V_1$, $C[M_2] \Downarrow V_2$.

From syntactic type safety, it is clear that mutual error approximation implies equivalence:

Lemma 267. *If* $\Gamma \vDash M_1 \sqsubseteq^{ctx} M_2 : A; \Gamma_M$ *and* $\Gamma \vDash M_2 \sqsubseteq^{ctx} M_1 : A; \Gamma_M$, *then* $\Gamma \vDash M_1 \approx^{ctx} M_2 : A; \Gamma_M$

Proof. The first two cases are direct. For the third case, we know by type safety that there are only 3 possibilities for a closed term's behavior: $C[M_i] \Uparrow$, $C[M_i] \Downarrow V_i$ or $C[M_i] \Downarrow \mho$. If $C[M_1] \Downarrow \mho$, then it is not the case that $C[M_1] \Uparrow$ or $C[M_1] \Downarrow V_1$ but by the first two cases that means that it is not the case that $C[M_2] \Uparrow$ or $C[M_1] \Downarrow V$, so it must be the case that $C[M_2] \Downarrow \mho$. The opposite direction follows by symmetry. □

To prove the soundness of the logical relation with respect to error approximation, we first need to construct for each step-index n a world $w_p^\sim(n)$ to hold the invariants for the cases generated in the preamble of a whole program, a substitution $\delta_p^\sim$ to give the relational interpretation of γ_p, and a "binary" version γ_p^2 of the preamble substitution γ_p.

Definition 268 (Preamble World, Relational Substitution)**.**

$$\begin{aligned}
\eta_p^\sim(n) &= \emptyset \boxplus (\texttt{Bool}, \texttt{Bool}, \mathcal{V}_n^\sim[\![\texttt{Bool}]\!]\emptyset\emptyset) \boxplus (\mathrm{OSum} \times \mathrm{OSum}, \mathrm{OSum} \times \mathrm{OSum}, \mathcal{V}_n^\sim[\![? \times \\
&\quad \boxplus (U(\mathrm{OSum} \to F\mathrm{OSum}), U(\mathrm{OSum} \to F\mathrm{OSum}), \lfloor \mathcal{V}_n^\sim[\![? \to ?]\!]\emptyset\emptyset \rfloor) \\
&\quad \boxplus (\exists X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \exists X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \mathcal{V}_n^\sim[\![\exists^\nu X.?]\!]\emptyset\emptyset) \\
&\quad \boxplus (U(\forall X.\mathrm{Case}\ X \to F\mathrm{OSum}), \forall X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \mathcal{V}_n^\sim[\![\forall^\nu X.?]\!]\emptyset\emptyset) \\
w_p^\sim(n) &= (n, \Sigma_p, \Sigma_p, \eta_p^\sim(n)) \\
\delta_p^\sim &= \emptyset, \mathrm{Bool} \mapsto (\texttt{Bool}, \texttt{Bool}, \mathcal{V}^\sim[\![\texttt{Bool}]\!]\emptyset\emptyset), \\
&\quad \mathrm{Times} \mapsto (\mathrm{OSum} \times \mathrm{OSum}, \mathrm{OSum} \times \mathrm{OSum}, \mathcal{V}^\sim[\![? \times ?]\!]\emptyset\emptyset), \\
&\quad \mathrm{Fun} \mapsto (U(\mathrm{OSum} \to F\mathrm{OSum}), U(\mathrm{OSum} \to F\mathrm{OSum}), \mathcal{V}^\sim[\![? \to ?]\!]\emptyset\emptyset), \\
&\quad \mathrm{Ex} \mapsto (\exists X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \exists X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \mathcal{V}^\sim[\![\exists^\nu X.?]\!]\emptyset\emptyset), \\
&\quad \mathrm{All} \mapsto (U(\forall X.\mathrm{Case}\ X \to F\mathrm{OSum}), \forall X.U(\mathrm{Case}\ X \to F\mathrm{OSum}), \mathcal{V}^\sim[\![\forall^\nu X.?]\!]\emptyset\emptyset)
\end{aligned}$$

$$\gamma_p^2(x) = (\gamma_p(x), \gamma_p(x))$$

The crucial property is that these together satisfy $\mathcal{G}^\sim[\![\Gamma_p]\!]$:

Lemma 269 (Validity of Preamble World)**.** *For every n, $(w_p^\sim(n), \gamma_p^2, \delta_p^\sim) \in \mathcal{G}^\sim[\![\Gamma_p]\!]$.*

Proof. Clear by definition. □

Lemma 270.

$$\frac{\Gamma \vDash M_l \sqsubseteq M_2 \in A; \Gamma_M}{\Gamma \vDash M_l \sqsubseteq^{ctx} M_2 \in A; \Gamma_M}$$

Proof. Let C be an appropriately typed closing context. By the congruence Lemma 264 we know

$$\cdot \vDash C[M_1] \sqsubseteq C[M_2] \in B; \cdot$$

By Lemma 269, we know that

$$(w_p^\sim(n), [\![C[M_1]]\!][\gamma_p], [\![C[M_2]]\!][\gamma_p] \in \mathcal{E}^\sim[\![B^\sqsubseteq]\!]\gamma_p^2\delta_p^\sim$$

(noting that $[\![C[M_i]]\!][\gamma_p][\delta_p^\sim] = [\![C[M_i]]\!][\gamma_p]$).

- If $C[M_1] \Uparrow$, then by the simulation Theorem 212 we know $\Sigma_p, [\![C[M_1]]\!][\gamma_p] \Uparrow$. Then, by adequacy for divergence Corollary 214, to show $C[M_2] \Uparrow$ it is sufficient to show that $\Sigma_p, [\![C[M_2]]\!][\gamma_p] \Uparrow$.

We will show that for every n, $\Sigma_p, [\![C[M_2]]\!][\gamma_p] \mapsto^n \Sigma_n, N_n$ for some Σ_n, N_n. We know

$$(w_p^{\succ}(n), [\![C[M_1]]\!][\gamma_p], [\![C[M_2]]\!][\gamma_p]) \in \mathcal{E}^{\succ}[\![B^{\sqsubseteq}]\!]\gamma\delta$$

we proceed by the cases of $\mathcal{E}^{\succ}[\![B^{\sqsubseteq}]\!]\gamma\delta$

- If $w_p^{\succ}(n).\Sigma_r, [\![[C[M_2]][\gamma_p]]\!] \mapsto^{w_p^{\succ}(n).j}$ we are done because $w_p^{\succ}(n).\Sigma_r = \Sigma_p$ and $w_p^{\succ}(n).j = n$.
- If $w_p^{\succ}(n).\Sigma_l, [\![[C[M_1]][\gamma_p]]\!] \mapsto^* \Sigma', \mathcal{U}$ we have a contradiction because $\Sigma_p, [\![[C[M_1]][\gamma_p]]\!] \Uparrow$
- If $w_p^{\succ}(n).\Sigma_l, [\![[C[M_1]][\gamma_p]]\!] \mapsto^* \Sigma', \texttt{ret}\ V_1$, we also have a contradiction for the same reason.

- If $C[M_1] \Downarrow V_1$ then by simulation we know $\Sigma_p, [\![C[M_1]]\!] \mapsto^n \Sigma', \texttt{ret}\ V_1'$ for some n, V_1. Furthermore, to show $C[M_2] \Downarrow V_2$ it is sufficient by simulation to show $\Sigma_p, [\![C[M_2]]\!] \mapsto^* \texttt{ret}\ V_2'$. We know

$$(w_p^{\prec}(n), [\![C[M_1]]\!][\gamma_p], [\![C[M_2]]\!][\gamma_p]) \in \mathcal{E}^{\prec}[\![B^{\sqsubseteq}]\!]\gamma\delta$$

we proceed by the cases of $\mathcal{E}^{\prec}[\![B^{\sqsubseteq}]\!]\gamma\delta$.

- If $w_p^{\prec}(n).\Sigma_l, [\![C[M_1]]\!][\gamma_p] \mapsto^{w_p^{\prec}(n).j+1}$ or $w_p^{\prec}(n).\Sigma_l, [\![C[M_1]]\!][\gamma_p] \mapsto^j \mathcal{U}$ this contradicts the fact that $\Sigma_p, [\![C[M_1]]\!] \mapsto^n \Sigma', \texttt{ret}\ V_1'$.
- Otherwise, $w_p^{\prec}(n).\Sigma_r, [\![C[M_2]]\!][\gamma_p] \mapsto^* \Sigma', \texttt{ret}\ V_2'$, so the result holds because $w_p^{\prec}(n).\Sigma_r = \Sigma_p$.

□

Finally we prove Lemma 264 that states that semantic term precision is a congruence.

Proof. By induction over C. In ever non-cast case we use the corresponding compatibility rule. The two casts cases are precisely dual.

- If $C = \langle A^{\sqsubseteq} \rangle\!\uparrow\ C'$, where $\Gamma, \Gamma_o' \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$ then we need to show

$$\Gamma' \vDash \langle A^{\sqsubseteq} \rangle\!\uparrow\ C'[M_l] \sqsubseteq \langle A^{\sqsubseteq} \rangle\!\uparrow\ C'[M_r] \in A_r; \Gamma_o'$$

First we use the upcast-left compatibility rule, meaning we need to show

$$\Gamma' \vDash C'[M_l] \sqsubseteq \langle A^{\sqsubseteq} \rangle\!\uparrow\ C'[M_r] \in A^{\sqsubseteq}; \Gamma_o'$$

Next, we use the upcast-right compatibility rule, meaning we need to show

$$\Gamma' \vDash C'[M_l] \sqsubseteq C'[M_r] \in A_l; \Gamma_o'$$

which follows by inductive hypothesis.

- If $C = \langle A^{\sqsubseteq} \rangle\!\downarrow C'$, where $\Gamma, \Gamma'_o \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r$ then we need to show

  $$\Gamma' \vDash \langle A^{\sqsubseteq} \rangle\!\downarrow C'[M_l] \sqsubseteq \langle A^{\sqsubseteq} \rangle\!\downarrow C'[M_r] \in A_l; \Gamma'_o$$

  First we use the downcast-right compatibility rule, meaning we need to show

  $$\Gamma' \vDash \langle A^{\sqsubseteq} \rangle\!\downarrow C'[M_l] \sqsubseteq C'[M_r] \in A^{\sqsubseteq}; \Gamma'_o$$

  Next, we use the downcast-left compatibility rule, meaning we need to show

  $$\Gamma' \vDash C'[M_l] \sqsubseteq C'[M_r] \in A^{\sqsubseteq}; \Gamma'_o$$

  which follows by inductive hypothesis.

□

10.7 PARAMETRICITY AND FREE THEOREMS

Our relational approach to proving the graduality theorem is not only elegant, it also makes the theorem more general, and in particular it *subsumes* the parametricity theorem that we want for the language, because we already assign arbitrary relations to abstract type variables. Then the parametricity theorem is just the reflexivity case of the graduality theorem.

Theorem 271 (Parametricity)**.** *If* $\Gamma \vdash M : A$*, then* $\Gamma \vDash M^+ \sqsubseteq M^+ : A$*.*

To demonstrate that this really is a parametricity theorem, we show that from this theorem we can prove "free theorems" that are true in polymorphic languages. These free theorems are naturally stated in terms of *contextual equivalence*, the gold standard for operational equivalence of programs, which we define as both programs diverging, erroring, or terminating successfully when plugged into an arbitrary context.

To use our logical relation to prove contextual equivalence, we need the following lemma, which says that semantic term precision both ways is *sound* for PolyG^v contextual equivalence.

Lemma 272. *If* $\Gamma \vDash M_l \sqsubseteq M_2 \in A$ *and* $\Gamma \vDash M_2 \sqsubseteq M_1 \in A$*, then* $\Gamma \vDash M_l \approx^{ctx} M_2 \in A; \Gamma_M$*.*

10.7.1 *Standard Free Theorems*

We now substantiate that this is a parametricity theorem by proving a few contextual equivalence results. First we present adaptations of some standard free theorems from typed languages, summarized in

$$\frac{M : \forall^{\nu} X.X \to X \qquad V_A : A \qquad V_B : B}{\texttt{let } f = M\{X \cong A\}; \texttt{unseal}_X(f\ \texttt{seal}_X V_A) \approx^{\text{ctx}} \texttt{let } f = M\{X \cong B\}; \texttt{let } y = (f\ \texttt{seal}_X V_B); V_A}$$

$$\frac{M : \forall^{\nu} X.\forall^{\nu} Y.(X \times Y) \to (Y \times X) \qquad V_A : A \qquad V_B : B}{\begin{array}{ll} \texttt{let } f = M\{X \cong A\}; & \texttt{let } f = M\{X \cong B\}; \\ \texttt{let } g = f\{Y \cong B\}; & \texttt{let } g = f\{Y \cong A\}; \\ \texttt{let } (y,x) = g\ (\texttt{seal}_X V_A)\ (\texttt{seal}_Y V_B); & \texttt{let } (y,x) = g\ (\texttt{seal}_X V_B)\ (\texttt{seal}_Y V_A); \\ (\texttt{unseal}_X x, \texttt{unseal}_Y y) & (\texttt{unseal}_Y y, \texttt{unseal}_X x) \end{array} \quad \approx^{\text{ctx}}}$$

$$\frac{\begin{array}{c} \text{NOT} = \lambda b : \texttt{Bool}.\texttt{if } b \texttt{ then false else true} \\ \text{WRAPNOT} = \lambda x : X.\texttt{seal}_X(\text{NOT}(\texttt{unseal}_X x)) \end{array}}{\begin{array}{c} \texttt{pack}^{\nu}(X \cong \texttt{Bool}, (\texttt{seal}_X \texttt{true}, (\text{WRAPNOT}, \lambda x : X.\texttt{unseal}_X x))) \\ \approx^{\text{ctx}} \texttt{pack}^{\nu}(X \cong \texttt{Bool}, (\texttt{seal}_X \texttt{false}, (\text{WRAPNOT}, \lambda x : X.\text{NOT}\,(\texttt{unseal}_X x)))) \end{array}}$$

Figure 10.29: Free Theorems without ?

Figure 10.29. The first equivalence shows that the behavior of any term with the "identity function type" $\forall^{\nu} X.X \to X$ must be independent of the input it is given. We place a λ on each side to delimit the scope of the X outward. Without the X (or a similar thunking feature like $\forall^{\nu}$ or $\exists^{\nu}$), the two programs would not have the same (effect) typing. In a more realistic language, this corresponds to wrapping each side in a module boundary. The next result shows that a function $\forall^{\nu} X.\forall^{\nu} Y.(X \times Y) \to (Y \times X)$, if it terminates, must flip the values of the pair, and furthermore whether it terminates, diverges or errors does not depend on the input values. Finally, we give an example using existential types. That shows that an abstract "flipper" which uses `true` for on and `false` for off in its internal state is equivalent to one using `false` and `true`, respectively as long as they return the same value in their "readout" function.

Theorem 273. *Given* $M : \forall^{\nu} X.X \to X, V_A : A, V_B : B,$

$$\texttt{let } f = M\{X \cong A\}; \texttt{unseal}_X(f\ \texttt{seal}_X V_A)$$

$$\approx^{ctx}$$

$$\texttt{let } f = M\{X \cong B\}; \texttt{let } y = (f\ \texttt{seal}_X V_B); V_A$$

Proof. There are 4 cases: $\sqsubseteq\prec, \sqsubseteq\succ, \succ\sqsupseteq$ and $\prec\sqsupseteq$ but they are all by a similar argument. Let $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_p, \Gamma]\!]$. We need to show

$$\begin{array}{l} (w, \\ \quad f \leftarrow [\![M]\!][\gamma_l][\delta_l]; \texttt{newcase}_{[\![A]\!]}\ c_X; f \leftarrow \texttt{force}\ f\ [[\![A]\!]]\ c_X; \\ \quad\quad [\![\texttt{unseal}_X(f\ \texttt{seal}_X V_A)]\!][\gamma_l][\delta_l], \\ \quad f \leftarrow [\![M]\!][\gamma_r][\delta_r]; \texttt{newcase}_{[\![B]\!]}\ c_X; f \leftarrow \texttt{force}\ f\ [[\![B]\!]]\ c_X; \\ \quad\quad [\![\texttt{let } y = (f\ \texttt{seal}_X V_B); V_A]\!][\gamma_r][\delta_r]) \\ \in \mathcal{E}^{\sim}[\![A]\!]\gamma\delta \end{array}$$

By the fundamental property, $(w, [\![M]\!][\gamma_l][\delta_r], [\![M]\!][\gamma_r][\delta_r])$. Applying monadic bind, we get $w' \sqsupseteq w$ with $(w', V_l, V_r) \in \mathcal{V}^{\sim}[\![\forall^{\nu} X.X \to X]\!]\gamma\delta$. By anti-reduction the goal reduces to

$$\begin{aligned}&(w,\\ &\texttt{newcase}_{[\![A]\!]}\ c_X; f \leftarrow \texttt{force}\ V_l\ [[\![A]\!]]\ c_X; [\![\texttt{unseal}_X(f\ \texttt{seal}_X V_A)]\!][\gamma_l][\delta_l],\\ &\texttt{newcase}_{[\![B]\!]}\ c_X; f \leftarrow \texttt{force}\ V_r\ [[\![B]\!]]\ c_X; [\![\texttt{let}\ y = (f\ \texttt{seal}_X V_B); V_A]\!][\gamma_r][\delta_r])\\ &\in \mathcal{E}^{\sim}[\![A]\!]\gamma\delta\end{aligned}$$

Next, each side allocates a new case and we need to pick a relation with which to instantiate it. By Lemmas 204 and 194, we have that $[\![V_A]\!][\gamma_l][\delta_l] \mapsto^0 \texttt{ret}\ V_{Al}$ and $[\![V_B]\!][\gamma_r][\delta_r] \mapsto^0 \texttt{ret}\ V_{Br}$ for some V_{Al} and V_{Br}. We define R to be the "singleton" relation:

$$R = \{(w, V_{Al}, V_{Br}) \in \mathrm{Atom}[[\![A]\!][\delta_l], [\![B]\!][\delta_r]] \mid w \sqsupseteq w'\}$$

Then we define w'' to be the world extended with $\lfloor R \rfloor_{w'.j}$:

$$w'' = (w'.j, w'.\Sigma_l, [\![A]\!][\delta_l], w'.\Sigma_r[\delta_r], w'.\eta \boxplus (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$$

where $\sigma_i = w'.\Sigma_i.size$.

Then clearly $w'' \sqsupseteq w'$ and after reducing some administrative redexes, the goal reduces to

$$(w'', S_l[\texttt{force}\ V_l[[\![A]\!]]\ \sigma_l], S_r[\texttt{force}\ V_r[[\![B]\!]]\ \sigma_r]) \in \mathcal{E}^{\sim}[\![A]\!]\gamma\delta$$

where

$$\begin{aligned}S_l = {} & f \leftarrow \bullet;\\ & x \leftarrow [\![V_A]\!][\gamma_l][\delta_l];\\ & \texttt{force}\ f\ x\end{aligned}$$

and

$$S_r = y \leftarrow \begin{pmatrix} \\ f \leftarrow \bullet; \\ x \leftarrow [\![V_B]\!][\gamma_r][\delta_r]; \\ \texttt{force}\ f\ x \end{pmatrix}; [\![V_A]\!][\gamma_r][\delta_r]$$

By definition, we have $(w'', S_l[\texttt{force}\ V_l[[\![A]\!]]\ \sigma_l], S_r[\texttt{force}\ V_r[[\![B]\!]]\ \sigma_r]) \in \mathcal{E}^{\sim}[\![X \to X]\!]\gamma'\delta'$ where $\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)$ and $\delta' = \delta, X \mapsto ([\![A]\!], [\![B]\!], R)$. Then, because by weakening $\mathcal{E}^{\sim}[\![A]\!]\gamma\delta = \mathcal{E}^{\sim}[\![A]\!]\gamma'\delta'$, we can apply monadic bind. Let $w''' \sqsupseteq w''$ and $(w''', V_f, V_f) \in \mathcal{V}^{\sim}[\![X \to X]\!]\gamma\delta$. After anti-reduction we need to show,

$$(w''', \texttt{force}\ V_{lf}\ V_{lA}, y \leftarrow (\texttt{force}\ V_{rf}\ V_{rB}); V_A[\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![A]\!]\gamma'\delta'$$

Since by definition we have $(w''', V_{lA}, V_{rB}) \in R = \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$, we get that $(w''', \texttt{force}\ V_{lf}\ V_{lA}, \texttt{force}\ V_{rf}\ V_{rB}) \in \mathcal{E}^{\sim}[\![X]\!]\gamma'\delta'$. We apply

monadic bind a final time, getting $w'''' \sqsupseteq w'''$ and needing to show (using the definition of R) that

$$(w'''', \texttt{ret}\ V_{lA}, y \leftarrow \texttt{ret}\ V_{rB}; V_A[\gamma_r][\delta_r]) \in \mathcal{E}^{\sim}[\![A]\!]\gamma'\delta'$$

which follows by reducing and finally the fundamental property for V_A. □

Theorem 274.

$$M : \forall^{\nu} X.\forall^{\nu} Y.(X \times Y) \to (Y \times X) \qquad V_A : A \qquad V_B : B$$

$$\begin{array}{l} \texttt{let}\ f = M\{X \cong A\}; \\ \texttt{let}\ g = f\{Y \cong B\}; \\ \texttt{let}\ (y, x) = g\ (\texttt{seal}_X V_A)\ (\texttt{seal}_Y V_B); \\ (\texttt{unseal}_X x, \texttt{unseal}_Y y) \end{array} \quad \approx^{ctx} \quad \begin{array}{l} \texttt{let}\ f = M\{X \cong B\}; \\ \texttt{let}\ g = f\{Y \cong A\}; \\ \texttt{let}\ (y, x) = g\ (\texttt{seal}_X V_B)\ (\texttt{seal}_Y V_A); \\ (\texttt{unseal}_Y y, \texttt{unseal}_X x) \end{array}$$

Proof. We show the $\sqsubseteq\sim$ case, the $\sim\sqsupseteq$ case is symmetric. Let $(w_0, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_p, \Gamma]\!]$. First, by Lemmas 204 and 194, we have that $[\![V_A]\!][\gamma_i][\delta_i] \mapsto^0 \texttt{ret}\ V_{Ai}$ and $[\![V_B]\!][\gamma_i][\delta_i] \mapsto^0 \texttt{ret}\ V_{Bi}$ for some V_{Ai} and V_{Bi} and that by the fundamental property $(w_0, V_{Al}, V_{Ar}) \in \mathcal{V}^{\sim}[\![A]\!]\gamma\delta$ and $(w_0, V_{Bl}, V_{Br}) \in \mathcal{V}^{\sim}[\![B]\!]\gamma\delta$.

Next, define

$$\begin{aligned} A_i &= [\![A]\!][\delta_i] \\ B_i &= [\![B]\!][\delta_i] \\ R_X &= \{(w, V_{Al}, V_{Br}) \in \mathrm{Atom}[A_l, B_r] \mid w \sqsupseteq w_0\} \\ R_Y &= \{(w, V_{Bl}, V_{Ar}) \in \mathrm{Atom}[B_l, A_r] \mid w \sqsupseteq w_0\} \end{aligned}$$

We need to show that

$$\begin{array}{l} (w_0, \\ f \leftarrow [\![M]\!][\gamma_l][\delta_l]; \texttt{newcase}_{A_l}\ c_X; \\ f \leftarrow \texttt{force}\ f\ [A_l]\ c_X; \\ f \leftarrow \texttt{ret}\ f; \texttt{newcase}_{B_l}\ c_Y; \\ g \leftarrow \texttt{force}\ f\ [B_l]\ c_Y; \\ [\![\texttt{let}\ (y, x) = g\ (\texttt{seal}_X V_A)\ (\texttt{seal}_Y V_B); (\texttt{unseal}_X x, \texttt{unseal}_Y y)]\!][\gamma_l][\delta_l], \\ f \leftarrow [\![M]\!][\gamma_r][\delta_r]; \texttt{newcase}_{B_r}\ c_X; \\ f \leftarrow \texttt{force}\ f\ [B_r]\ c_X; \\ f \leftarrow \texttt{ret}\ f; \texttt{newcase}_{A_r}\ c_Y; \\ g \leftarrow \texttt{force}\ f\ [A_r]\ c_Y; \\ [\![\texttt{let}\ (y, x) = g\ (\texttt{seal}_X V_B)\ (\texttt{seal}_Y V_A); (\texttt{unseal}_Y y, \texttt{unseal}_X x)]\!][\gamma_r][\delta_r]) \\ \in \mathcal{E}^{\sim}[\![A \times B]\!]\gamma\delta \end{array}$$

Now, the translation of each term has $[\![M]\!][\gamma_i][\delta_i]$ in evaluation position so we can apply monadic bind, giving us some $w_1 \sqsupseteq w_0$ and $(w_1, V_{Ml}, V_{Mr}) \in \mathcal{V}^{\sim}[\![\forall^{\nu} X.\forall^{\nu} Y.X \times Y \to Y \times X]\!]\gamma\delta$. Next, define

$$\begin{aligned}
w_2 = \; & (w_1.j, \\
& (w_1.\Sigma_l, A_l, B_l), (w_1.\Sigma_r, B_r, A_r), \\
& (w_1.\eta \boxplus (A_l, B_r, \lfloor R_X \rfloor_{w_1.j}) \boxplus (B_l, A_r, \lfloor R_Y \rfloor_{w_1.j})))
\end{aligned}$$

$$\begin{aligned}
\sigma_{Xi} &= w_1.\Sigma_i.size \\
\sigma_{Yi} &= w_1.\Sigma_i.size + 1 \\
\gamma' &= \gamma, c_X \mapsto (\sigma_{Xl}, \sigma_{Xr}), c_Y \mapsto (\sigma_{Yl}, \sigma_{Yr}) \\
\delta' &= \delta, X \mapsto (A_l, B_r, R_X), Y \mapsto (B_l, A_r, R_Y)
\end{aligned}$$

Then we can apply monadic bind a few times and get some $w_3 \sqsupseteq w_2$ with $(w_3, V_{fl}, V_{fr}) \in \mathcal{V}^{\sim}[\![X \times Y \to Y \times X]\!]\gamma'\delta'$ and we need to show that

$$\begin{aligned}
& (w_3, \\
& [\![\texttt{let}\ (y,x) = g\ (\texttt{seal}_X V_A)\ (\texttt{seal}_Y V_B); (\texttt{unseal}_X x, \texttt{unseal}_Y y)]\!][\gamma_l''][\delta_l'], \\
& [\![\texttt{let}\ (y,x) = g\ (\texttt{seal}_X V_B)\ (\texttt{seal}_Y V_A); (\texttt{unseal}_Y y, \texttt{unseal}_X x)]\!][\gamma_r''][\delta_r']) \\
& \in \mathcal{E}^{\sim}[\![A \times B]\!]\gamma'\delta'
\end{aligned}$$

where $\gamma'' = \gamma', g \mapsto (V_{fl}, V_{fr})$. By anti-reduction the goal reduces to

$$(w_3, p \leftarrow \texttt{force}\ V_{fl}\ V_{Al}\ V_{Bl}; \texttt{let}\ (y,x) =$$

$$p; [\![(\texttt{unseal}_X x, \texttt{unseal}_Y y)]\!][\delta_l'], p \leftarrow \texttt{force}\ V_{fr}\ V_{Br}\ V_{Ar}; \texttt{let}\ (y,x) =$$

$$p; [\![(\texttt{unseal}_Y y, \texttt{unseal}_X x)]\!][\delta_r'])\mathcal{E}^{\sim}[\![A \times B]\!]\gamma'\delta'$$

Then we use monadic bind combined with the fact that based on the relations R_X, R_Y we know $(w_3, \texttt{force}\ V_{fl}\ V_{Al}\ V_{Bl}, \texttt{force}\ V_{fr}\ V_{Br}\ V_{Ar}) \in \mathcal{E}^{\sim}[\![X \times Y]\!]\gamma'\delta'$. We get one last $w_4 \sqsupseteq w_3$ and the goal finally reduces to (simplifying using R_X, R_Y)

$$(w_4, \texttt{ret}\ (V_{Al}, V_{Bl}), \texttt{ret}\ (V_{Ar}, V_{Br})) \in \mathcal{E}^{\sim}[\![A \times B]\!]\gamma'\delta'$$

which follows by our earlier assumption by weakening. □

Lemma 275.

$$\begin{aligned}
& \mathit{pack}^{\nu}(X \cong \mathtt{Bool}, (\mathit{seal}_X \mathtt{true}, (\mathit{NOT}, \lambda x : X.\mathit{unseal}_X x))) \\
\approx^{ctx}\ & \mathit{pack}^{\nu}(X \cong \mathtt{Bool}, (\mathit{seal}_X \mathtt{false}, (\mathit{NOT}, \lambda x : X.\mathit{NOT}\,(\mathit{unseal}_X x))))
\end{aligned}$$

Proof. We do the $\sqsubseteq\sim$ case, the $\sim\sqsupseteq$ case is symmetric. Let $(w, \gamma, \delta) \in \mathcal{G}^{\sim}[\![\Gamma_p]\!]$. The goal reduces to showing

$$\begin{aligned}
& \left(w, \begin{array}{l} \texttt{pack}(\texttt{Bool}, (\texttt{thunk}\ \lambda c_X : \text{Case}\ X.[\![(\texttt{seal}_X \texttt{true}, (\text{NOT}, (\lambda x : X.\texttt{unseal}_X x)))]\!])), \\ \texttt{pack}(\texttt{Bool}, \texttt{thunk}\ \lambda c_X : \text{Case}\ X.[\![(\texttt{seal}_X \texttt{false}, (\text{NOT}, \lambda x : X.\text{NOT}\,(\texttt{unseal}_X x)))]\!]) \end{array} \right) \\
& \in \mathcal{V}^{\sim}[\![\exists^{\nu} X.X \times ((X \to X) \times (X \to \texttt{Bool}))]\!]\gamma\delta
\end{aligned}$$

The relation we pick is $R = \{(w', \texttt{true}, \texttt{false}) \in \text{Atom}[\texttt{Bool}, \texttt{Bool}]\} \cup \{(w', \texttt{false}, \texttt{true}) \in \text{Atom}[\texttt{Bool}, \texttt{Bool}]\}$ Then we need to show for any future $w' \sqsupseteq w$ and $w' \vDash (\sigma_l, \sigma_r, \lfloor R \rfloor_{w'.j})$, that

$$\left(w', \begin{array}{l}(\texttt{force}\ (\texttt{thunk}\ \lambda c_X : \text{Case}\ X.[\![(\texttt{seal}_X\texttt{true}, (\text{NOT}, \lambda x : X.\texttt{unseal}_X x))]\!])\ \sigma_l), \\ (\texttt{force}\ \texttt{thunk}\ \lambda c_X : \text{Case}\ X.[\![(\texttt{seal}_X\texttt{true}, (\text{NOT}, \lambda x : X.\text{NOT}\ (\texttt{unseal}_X x)))]\!]\ \sigma_r)\end{array}\right)$$
$$\in \mathcal{E}^{\sim}[\![X \times ((X \to X) \times (X \to \texttt{Bool}))]\!]\gamma'\delta'$$

where

$$\gamma' = \gamma, c_X \mapsto (\sigma_l, \sigma_r)$$
$$\delta' = \delta, X \mapsto (\texttt{Bool}, \texttt{Bool}, R)$$

And after applying anti-reduction, by monadic bind, we need to show the following 3 things:

$$(w', \texttt{true}, \texttt{false}) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$$

$$(w', [\![\text{NOT}]\!][\gamma'_l][\delta'_l], [\![\text{NOT}[\gamma'_r][\delta'_r]]\!]) \in \mathcal{E}^{\sim}[\![X \to X]\!]\gamma'\delta'$$

$$(w', [\![\lambda x : X.\texttt{unseal}_X x]\!], [\![\lambda x : X.\text{NOT}\ (\texttt{unseal}_X x)]\!]) \in \mathcal{E}^{\sim}[\![X \to \texttt{Bool}]\!]\gamma'\delta'$$

1. First, $(w', \texttt{true}, \texttt{false}) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$ follows directly from the definition of $\delta_R(X) = R$.

2. Second, let $w'' \sqsupseteq w'$ and $(w'', V_l, V_r) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$. Then we need to show

   $$(w'', \texttt{force}\ V_{NOT}[\gamma'_l][\delta'_l]\ V_l, \texttt{force}\ V_{NOT}[\gamma'_r][\delta'_r]\ V_r) \in \mathcal{E}^{\sim}[\![X]\!]\gamma'\delta'$$

   where $[\![\text{NOT}]\!] = \texttt{ret}\ V_{NOT}$. There are two cases: either $V_l = \texttt{true}$ and $V_r = \texttt{false}$ or vice-versa. In either case, NOT swaps the two values and the result holds.

3. Finally, let $w'' \sqsupseteq w'$ and $(w'', V_l, V_r) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$. Then we need to show,

   $$(w'', \texttt{force}\ V_{fl}\ V_l, \texttt{force}\ V_{fr}\ V_r) \in \mathcal{E}^{\sim}[\![\texttt{Bool}]\!]\gamma'\delta'$$

   where $[\![\lambda x : X.\texttt{unseal}_X x]\!] = \texttt{ret}\ V_{fl}$ and $[\![\lambda x : X.\text{NOT}\ (\texttt{unseal}_X x)]\!] = \texttt{ret}\ V_{fr}$. By definition of R, either $V_l = \texttt{true}$ and $V_r = \texttt{false}$ or vice-versa. In either case, both sides evaluate to $\texttt{ret}\ V_l$, and we need to show

   $$(w'', V_l, V_l) \in \mathcal{V}^{\sim}[\![\texttt{Bool}]\!]\gamma'\delta'$$

   which follows by definition.

□

10.7.2 *Free Theorems with Dynamic Typing*

Next, to give a flavor of what kind of relational reasoning is possible in the presence of the dynamic type, we consider what free theorems are derivable for functions of type $\forall X.? \to X$. A good intuition for this type is that the only possible outputs of the function are sealed values that are contained within the dynamically typed input. It is difficult to summarize this in a single statement, so instead we give the following three examples:

Theorem 276 ($\forall^v X.? \to X$ Free Theorems)**.** *Let* $\cdot \vdash M : \forall^v X.? \to X$

1. *For any* $\cdot \vdash V : ?$, *then* $\texttt{let}\ f = M\{X \cong A\}; \texttt{unseal}_{f\,V}$ *either diverges or errors.*
2. *For any* $\cdot \vdash V : A$ *and* $\cdot \vdash V' : B$,

$$\begin{array}{l}\texttt{let}\ f = M\{X \cong A\};\\ \texttt{unseal}_X(f\ \texttt{seal}_X V)\end{array} \approx^{ctx} \begin{array}{l}\texttt{let}\ f = M\{X \cong A\};\\ \texttt{let}\ y = (\texttt{unseal}_X(f\ \texttt{seal}_X V'));\\ V\end{array}$$

3. *For any* $\cdot \vdash V : A, \cdot \vdash V' : B, \cdot \vdash V_d : ?$,

$$\begin{array}{l}\texttt{let}\ f = M\{X \cong A\};\\ \texttt{unseal}_X(f\ (\texttt{seal}_X V, V_d))\end{array} \approx^{ctx} \begin{array}{l}\texttt{let}\ f = M\{X \cong B\};\\ \texttt{let}\ y = \texttt{unseal}_X(f\ (\texttt{seal}_X V', V_d));\\ V\end{array}$$

The first example passes in a value V that does not use the seal X, so we know that the function cannot possibly return a value of type X. The second example mimics the identity function's free theorem. It passes in a sealed value V and the equivalence shows that V''s effects do not depend on what V was sealed and the only value that V' can return is the one that was passed in. The third example illustrates that there are complicated ways in which sealed values might be passed as a part of a dynamically typed value, but the principle remains the same: since there is only one sealed value that's part of the larger dynamically typed value, it is the only possible return value, and the effects cannot depend on its actual value. The proof of the first case uses the relational interpretation that X is empty. The latter two use the interpretation that X includes a single value.

Compare this reasoning to what is available in GSF, where the polymorphic *function* determines which inputs are sealed and which are not, rather than the caller. Because of this, Toro, Labrada, and Tanter [85] only prove "cheap" theorems involving ? where the polymorphic function is known to be a literal Λ function and not a casted function. As an example, for arbitrary $M : \forall X.? \to X$, consider the application $M\,[\texttt{Bool}](\texttt{true}, \texttt{false})$. The continuation of this call has no way of

knowing if neither, one or both booleans are members of the abstract type X. The following examples of possible terms for M illustrate these three cases:

1. $M_1 = (\Lambda X.\lambda x : \texttt{Bool} \times \texttt{Bool}.\texttt{if or}\, x\, \texttt{then}\, \mho\, \texttt{else}\, \Omega) :: \forall X.? \to X$
2. $M_2 = (\Lambda X.\lambda x : X \times \texttt{Bool}.\texttt{if}\, \mathrm{snd}\, x\, \texttt{then}\, \mathrm{fst}\, x\, \texttt{else}\, \Omega) :: \forall X.? \to X$
3. $M_3 = (\Lambda X.\lambda x : X \times X.\mathrm{snd}\, x) :: \forall X.? \to X$

If $M = M_1$, both booleans are concrete so X is empty, but from the inputs the function can determine whether to diverge or error. If $M = M_2$, the first boolean is abstract and the second is concrete, so only the first can inhabit X, but the second can be used to determine whether to return a value or not. Finally if $M = M_3$, both booleans are abstract so the function cannot inspect them, but either can be returned. It is unclear what reasoning the continuation has here: it must anticipate every possible way in which the function might decide which values to seal, and so has to consider every dynamically typed value of the instantiating type as possibly abstract and possibly concrete. Most of the free theorems are stated in terms of contextual equivalence. We define contextual equivalence of PolyG^v terms to mean contextual equivalence of their elaborations into PolyC^v terms. By Lemma 267, we can prove a contextual equivalence $M_1 \approx^{\mathrm{ctx}} M_2$ by proving contextual error approximation both ways ($M_1 \sqsubseteq^{ctx} M_2$ and $M_2 \sqsubseteq^{ctx} M_1$). We can prove contextual error approximation by proving logical relatedness by soundness of the logical relation for open terms (Lemma 270), which is defined in terms of the two logical relations $\sqsubseteq\prec, \sqsubseteq\succ$, giving us technically 4 things to prove: $\sqsubseteq\prec, \sqsubseteq\succ, \succ\sqsupseteq$ and $\prec\sqsupseteq$. However, these cases are all very similar so we show only one case and the other cases follow by essentially symmetric arguments.

Now we give formal proofs of the three cases.

Theorem 277. *Let $M_f : \forall^v X.? \to X$ and $\cdot \vdash A$ and $\cdot \vdash M_d : ?$ and $M_l, M_r : B$, be closed well-typed terms and define*

$$N_i = \texttt{let}\ f = M_f\{X \cong A\}; \texttt{let}\ y = f\ M_d; M_i$$

Then

$$N_1 \approx^{ctx} N_2$$

And therefore, for any M_i, $N_i \Uparrow$ or $N_i \mapsto^* \mho$.

Proof. First, note that the contextual equivalence is clearly false if N_i reduces to a value since we can pick $M_l = \texttt{true}$ and $M_r = \texttt{false}$, so N_i must diverge or error.

The sketch of the proof is as follows. We use the empty relational interpretation for X. M_d is well-behaved with this interpretation since

X is not used (weakening). So if we pass M_d's value to M we get a well-behaved X term out, but there are no terminating well-behaved X terms since the interpretation of X is empty. Therefore the M_l, M_r are dead code and the terms are equivalent.

Now to the in-depth formal proof. By symmetry it is sufficient to prove for any $(w, \gamma, \delta) \in \mathcal{G}^\sim[\![\Gamma_p]\!]$ that $(w, [\![N_l^+]\!][\gamma_l][\delta_l], [\![N_r^+]\!][\gamma_l][\delta_l]) \in \mathcal{E}^\sim[\![B]\!]\gamma\delta$.

Expanding definitions, define

$$\begin{aligned}
M'_{fi} &= [\![M_f^+]\!][\gamma_i][\delta_i] \\
M'_{di} &= [\![M_d^+]\!][\gamma_i][\delta_i] \\
M'_i &= [\![M_i^+]\!][\gamma_i][\delta_i] \\
A'_i &= [\![A]\!][\delta_i] \\
N'_i &= \begin{array}{l} f_p \leftarrow M'_{fi}; \texttt{newcase}_{A'_i}\ c_X; f \leftarrow \texttt{force}\ f_p\ [A'_i]\ c_X; \\ y \leftarrow (f \leftarrow \texttt{ret}\ f; x \leftarrow M'_{di}; \texttt{force}\ f\ x); M'_i \end{array}
\end{aligned}$$

Then in these terms, our goal is to show $(w, N'_l, N'_r) \in \mathcal{E}^\sim[\![B]\!]\gamma\delta$. First, $(w_0, M_{fl'}, M'_{fr}) \in \mathcal{V}^\sim[\![\forall^\nu X.? \to X]\!]\gamma\delta$, so we can apply monadic bind. Let $w_1 \sqsupseteq w_0$ and $(w_1, V'_{fl}, V'_{fr}) \in \mathcal{V}^\sim[\![B]\!]\gamma\delta$. Then, define

$$N'_{i1} = \begin{array}{l} \texttt{newcase}_{A'_i}\ c_X; f \leftarrow \texttt{force}\ V'_{fi}\ [A'_i]\ c_X; \\ y \leftarrow (f \leftarrow \texttt{ret}\ f; x \leftarrow M'_{di}; \texttt{force}\ f\ x); M'_i \end{array}$$

And we need to show $(w_1, N'_{l1}, N'_{l2}) \in \mathcal{E}^\sim[\![B]\!]\gamma\delta$. Define

$$\begin{aligned}
R_\varnothing &= \varnothing \\
w_2 &= (w_1.j, (w_1.\Sigma_l, A'_l), (w_1.\Sigma_r, A'_r), (w_1.\eta \boxplus (A_l, A_r, \lfloor R_\varnothing \rfloor_{w_1.j}))) \\
\sigma_{Xi} &= w_1.\Sigma_i.size \\
\gamma' &= \gamma, c_X \mapsto (\sigma_l, \sigma_r) \\
\delta' &= \delta, X \mapsto (\texttt{Bool}, \texttt{Bool}, R)
\end{aligned}$$

Then both sides reduce, allocating a fresh tag. Define

$$N'_{i2} = \begin{array}{l} f \leftarrow \texttt{force}\ V'_{fi}\ [A'_i]\ \sigma_i; \\ y \leftarrow (f \leftarrow \texttt{ret}\ f; x \leftarrow M'_{di}; \texttt{force}\ f\ x); M'_i \end{array}$$

Then it is sufficient to prove $(w_2, N'_{l2}, N'_{r2}) \in \mathcal{E}^\sim[\![B]\!]\gamma\delta$. Note that by weakening, $\mathcal{E}^\sim[\![B]\!]\gamma\delta = \mathcal{E}^\sim[\![B]\!]\gamma'\delta'$, so we will prove $(w_2, N'_{l2}, N'_{r2}) \in \mathcal{E}^\sim[\![B]\!]\gamma\delta$.

Next, by definition of the logical relation $(w_2, \texttt{force}\ V'_{fl}\ [A'_l]\ \sigma_l, \texttt{force}\ V'_{fr}\ [A'_r]\ \sigma_r) \in \mathcal{V}^\sim[\![? \to X]\!]\gamma'\delta'$, so we can again use monadic bind. Let $w_3 \sqsupseteq w_2$ and $(w_3, V''_{fl}, V''_{fr}) \in \mathcal{V}^\sim[\![? \to X]\!]\gamma'\delta'$. Reducing we get

$$N'_{i3} = \ y \leftarrow (x \leftarrow M'_{di}; \texttt{force}\ V''_{fi}\ x); M'_i$$

And the goal is to prove $(w_3, N'_{l3}, N'_{r3}) \in \mathcal{E}^\sim[\![B]\!]\gamma'\delta'$.

To apply monadic bind, we need to show that $(w_3, M'_{dl}, M'_{dr}) \in \mathcal{V}^\sim[\![?]\!]\gamma'\delta'$, which follows by the fundamental property So we apply monadic bind again, and get $w_4 \sqsupseteq w_3$ and $(w_4, V'_{dl}, V'_{dr}) \in \mathcal{V}^\sim[\![?]\!]\gamma'\delta'$. Reducing, define

Note that it is key here that X is free in M_d, otherwise we would not get that these terms are related with X interpreted as $R_\emptyset$, but only X interpreted as $\mathcal{V}^\sim[\![A]\!]$.

$$N'_{i4} = y \leftarrow (\texttt{force}\ V''_{fi}\ V'_{di}); M'_i$$

And we need to show $(w_4, N'_{l4}, N'_{r4}) \in \mathcal{E}^\sim[\![B]\!]\gamma'\delta'$. By definition,

$$(w_4, (\texttt{force}\ V''_{fl}\ V'_{dl}), (\texttt{force}\ V''_{fr}\ V'_{dr})) \in \mathcal{E}^\sim[\![X]\!]\gamma'\delta'$$

, so we can apply monadic bind. We get $w_5 \sqsupseteq w_4$ and $(w_5, V_{Xl}, V_{Xr}) \in \mathcal{V}^\sim[\![X]\!]\gamma'\delta'$. The goal at this point is irrelevant since $\mathcal{V}^\sim[\![X]\!]\gamma'\delta' = R_\emptyset$, and so we have a contradiction. □

Theorem 278. *For any $M_f : \forall^\nu X.? \to X \cdot \vdash A, B$ and $V_A : A$ and $V_B : B$,*

$$\begin{array}{lcl} \texttt{let}\ f = M_f\{X \cong A\}; & & \texttt{let}\ f = M_f\{X \cong B\}; \\ \texttt{let}\ y = (f\ \texttt{seal}_X V_A); & \approx^{ctx} & \texttt{let}\ y = (f\ \texttt{seal}_X V_B); \\ \texttt{unseal}_X y & & V_A \end{array}$$

Proof. The sketch of the proof is as follows. As the relational interpretation of X we pick the singleton relation relating V and V'. Then $\texttt{seal}_X V$ and $\texttt{seal}_X V'$ are related values at the dynamic type ?. Then by parametricity of $M : \forall^\nu X.? \to X$, if M returns, it will return $\texttt{seal}_X V$ in the left term and the $\texttt{seal}_X V'$ in the right term.

We show one direction of the 4 cases, the others are analogous. Let $(w_0, \gamma, \delta) \in \mathcal{G}^\sim[\![\Gamma_p]\!]$. Define

$$\begin{array}{rl} & f_p \leftarrow M_{fl}; \texttt{newcase}_{A_l}\ c_X; f \leftarrow \texttt{force}\ f_p\ [A_l]\ c_X; \\ N_l = & y \leftarrow f \leftarrow \texttt{ret}\ f_p; x \leftarrow (a \leftarrow M_{VAl}; \texttt{ret}\ \texttt{inj}_{c_X}\ a); (\texttt{force}\ f\ a); \\ & \texttt{ret}\ y \end{array}$$

$$\begin{array}{rl} & f_p \leftarrow M_{fr}; \texttt{newcase}_{B_r}\ c_X; f \leftarrow \texttt{force}\ f_p\ [B_r]\ c_X; \\ N_r = & y \leftarrow f \leftarrow \texttt{ret}\ f_p; x \leftarrow (a \leftarrow M_{VBr}; \texttt{ret}\ \texttt{inj}_{c_X}\ a); (\texttt{force}\ f\ a); \\ & M_{VAr} \end{array}$$

$$\begin{aligned} M_{fi} &= [\![M_f^+]\!][\gamma_i][\delta_i] \\ M_{VAi} &= [\![V_A^+]\!][\gamma_i][\delta_i] \\ M_{VBi} &= [\![V_B^+]\!][\gamma_i][\delta_i] \\ A_i &= [\![A]\!][\delta_i] \\ B_i &= [\![B]\!][\delta_i] \end{aligned}$$

And we need to show $(w_0, N_l, N_r) \in \mathcal{E}^\sim[\![A]\!]\gamma\delta$.

First, by Lemmas 204 and 194, note that we know $M_{VAi} \mapsto^0 \texttt{ret}\ V_{Ai}$ and $M_{VBi} \mapsto^0 \texttt{ret}\ V_{Bi}$, with $(w_0, V_{Al}, V_{Ar}) \in \mathcal{V}^\sim[\![A]\!]\gamma\delta$ and $(w_0, V_{Bl}, V_{Br}) \in \mathcal{V}^\sim[\![A]\!]\gamma\delta$. Then define

$$R_X = \{(w, V_l, V_r) \mid V_l = V_{Al} \wedge V_r = V_{Ar} \wedge w \sqsupseteq w_0\}$$

Then we proceed as in the previous proof. Getting some $w_1 \sqsupseteq w_0$ that uses R_X as the interpretation of X. Define

$$N_{l1} = \begin{array}{l} y \leftarrow (\texttt{force}\ V_{fl}\ (\texttt{inj}_{\sigma_l}\ V_{Al})); \\ \texttt{ret}\ y \end{array}$$

$$N_{l1} = \begin{array}{l} y \leftarrow (\texttt{force}\ V_{fr}\ (\texttt{inj}_{\sigma_r}\ V_{Br})); \\ M_{VAr} \end{array}$$

Where $(w_1, V_{fl}, V_{fr}) \in \mathcal{V}^{\sim}[\![? \to X]\!]\gamma'\delta'$ and $\gamma', \delta', \sigma_i$ are defined analogously to the previous proof. By weakening our goal is equivalent to $(w_1, N_{l1}, N_{r1}) \in \mathcal{E}^{\sim}[\![A]\!]\gamma'\delta'$. Next, we can prove

$$(w_1, \texttt{force}\ V_{fl}\ (\texttt{inj}_{\sigma_l}\ V_{Al}), \texttt{force}\ V_{fr}\ (\texttt{inj}_{\sigma_r}\ V_{Br})) \in \mathcal{E}^{\sim}[\![X]\!]\gamma'\delta'$$

if we can show that $(w_1, \texttt{inj}_{\sigma_l}\ V_{Al}, \texttt{inj}_{\sigma_r}\ V_{Br}) \in \mathcal{V}^{\sim}[\![?]\!]\gamma'\delta'$. By definition of σ_l, σ_r this follows because $(w_1, V_{Al}, V_{Br}) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta' = R_X$. So we can apply monadic bind, getting some $w_2 \sqsupseteq w_1$ and $(w_2, V'_{Al}, V'_{Br}) \in \mathcal{V}^{\sim}[\![X]\!]\gamma'\delta'$. Note that necessarily by definition of R_X, that $V'_{Al} = V_{Al}$ and $V'_{Br} = V_{Br}$. So our goal then becomes to show (after anti-reduction) that

$$(w_2, \texttt{ret}\ V_{Al}, V_{Ar}) \in \mathcal{E}^{\sim}[\![A]\!]\gamma'\delta'$$

Which follows by weakening and the fundamental property. □

Theorem 279. *For any* $\cdot \vdash A, B, M : \forall^{\nu} X.? \to X$ *and* $V_A : A, V_B : B, \cdot \vdash V_d : ?$,

$$\texttt{let}\ f = M\{X \cong A\}; \texttt{unseal}_X(f\ (\texttt{seal}_X V_A, V_d)) \approx^{ctx} \texttt{let}\ f = M\{X \cong B\}; \texttt{let}\ y = f\ (\texttt{seal}_X V_B, V_d); V_A$$

Proof. Similar to the previous theorem, using the same relational interpretation. The key is to show that

$$(w', \texttt{inj}_{\text{case}(?\times?)}\ (\texttt{inj}_{\sigma_l}\ V_{Al}, V_{dl}), (\texttt{inj}_{\text{case}(?\times?)}\ (\texttt{inj}_{\sigma_r}\ V_{Br}), V_{dr})) \in \mathcal{V}^{\sim}[\![?]\!]\gamma'\delta'$$

where the relational interpretation of X in (w', γ', δ') is the singleton relating V_{Al} and V_{Br}. This follows by weakening and the rest of the argument follows by similar argument to the previous proof. □

10.8 DISCUSSION AND RELATED WORK

DYNAMICALLY TYPED POLYG$^{\nu}$ AND DESIGN ALTERNATIVES Most gradually typed languages are based on adding types to an existing dynamically typed language, with the static types capturing some feature already existing in the dynamic language that can be migrated

to use static typing. PolyG$^\nu$ was designed as a proof-of-concept standalone gradual language, so it might not be clear what dynamic typing features it supports migration of. In particular, since all sealing is explicit, PolyG$^\nu$ does not model migration from programming without seals entirely to programming with them, so its types are relevant to languages that include some kind of nominal data type generation, along the lines of Dynamically typed PolyG$^\nu$.

The existential types of dynamically typed PolyG$^\nu$ have some analogues in other languages. For example, in Racket structs can be used to make fresh nominal types and *units* provide first-class modules. It would be interesting future work to see if our logical relation can usefully be adapted to Typed Racket's typed units [83].

Our fresh polymorphic types are more exotic than the fresh existentials, and don't clearly correspond to any existing programming features, but they model abstraction over nominal datatypes where the datatype is guaranteed to be freshly generated.

Our use of abstract and known type variables was directly inspired by Neis, Dreyer, and Rossberg [55], who present a language with a fresh type creation mechanism which they show enables parametric reasoning though the language overall does not satisfy a traditional parametricity theorem. This suggests an alternative language design, where $\forall$ and $\exists$ behave normally and we add a `newtype` facility, analogous to that feature of Haskell, where `newtype` allocates a new case of the open sum type for each type it creates.

TAG CHECKING Siek et al. [75] claim that graduality demands that tag-checking functions like our `is(Bool)?` form must error when applied to sealed values, and used this as a criticism of the design of Typed Racket. However, in our language, `is(Bool)?` will simply return `false`, which matches Typed Racket's behavior. This is desirable if we are adding types to an existing dynamic language, because typically a runtime tag check should be a *safe* operation in a dynamically typed language. Explicit sealing avoids this graduality issue, an advantage over previous work.

LOGICAL RELATIONS Our use of explicit sealing eliminates much of the complexity of prior logical relations [5, 85]. To accommodate dynamic conversion and evidence insertion, those relations adopted complex value relations for universal types that in turn restricted the ways in which they could treat type variables. Additionally, we are the first to give a logical relation for fresh *existential* types, and it is not clear how to adapt the non-standard relation for universals to existentials [5, 85].

Next, while we argue that our logical relation more fully captures parametricity than previous work on gradual polymorphism, this is not a fully formal claim. To formalize it, in future work we could

show that PolyG$^{\nu}$ is a model of an effectful variant of an axiomatic parametricity formulation such as Dunphy [19], Ma and Reynolds [48], and Plotkin and Abadi [63].

NONINTERFERENCE Toro, Garcia, and Tanter [84] prove termination-insensitive noninterference, for a language with gradual security via a logical relations argument. Analogous to Toro, Labrada, and Tanter [85], applying the AGT approach produces a language that satisfies graduality but not noninterference, so they tweak the language to satisfy noninterference but not graduality and make similar conjectures that the combination of graduality and noninterference is impossible.

Later, after the publication of the paper this chapter is based on, a system was developed that provides non-interference and graduality, [6]. Their solution, developed independently, bears similarities to our approach: they separate the effect of gradual type ascriptions from the confidentiality of data, similar to how PolyG$^{\nu}$ separates sealing and unsealing from gradual enforcement.

Part IV

CONCLUSIONS

11 DISCUSSION AND FUTURE WORKS

11.1 IMPLEMENTATION AND OPTIMIZATION

In this dissertation, I have focused on the *semantics* of gradual typing. However, the most pressing issue with gradual typing is *performance*, specifically the high runtime space and time overheads of casts, as demonstrated in [35, 78]. While out of scope for this dissertation, it is of great interest to see if our approach will provide insights into gradual typing implementations.

I hope that our analysis in Chapter 5 on different cast semantics helps elucidate the tradeoff that is being made by attempts to make gradual typing more efficient by using alternative cast semantics such as transient [87] and amnesic [34] semantics. These semantics sacrifice the η laws to get more efficient implementations of their higher-order casts. By showing the deep connection between η laws and the wrapping semantics, I hope that we have shown why the goal of improving the performance of wrapping semantics in particular is a valuable pursuit.

As to what insights might lead to improved implementations, most directly relevant would be the thunkability of upcasts and linearity of downcasts that we identified in Chapter 5. These two properties are really the most important properties for an optimizer to know. Thunkability means a computation can be treated as a value, and so can be freely inlined, duplicated, de-duplicated, hoisted, etc. Similarly, linearity means *strictness* of a computation, which is crucial to optimization of lazy languages because it can justify the use of more efficient call-by-value calling-conventions.

11.2 WHAT DO TYPE AND TERM PRECISION MEAN?

Throughout this dissertation, we have approached the semantic study of gradual typing in a few concrete presentations, all roughly equivalent and all based on the idea that upcasts and downcasts form embedding projection pairs.

- In Chapter 3, we present a semantic interpretation saying that type precision $A \sqsubseteq B$ holds when there exists an embedding-projection pair between A and B that factorizes the ep pair that each canonically has with the dynamic type. Then term precision is defined by first defining a homogeneous error approximation relation, which is made heterogeneous by the insertion of casts.

- In Chapter 5, we axiomatize type and term precision as abstract relations, with the key property that each type precision $A \sqsubseteq B$ implies the existence of pure upcasts (for call-by-value) and linear downcasts (for call-by-name). When restricting to one evaluation order, we recover the adjointness of the casts, and under mild assumptions about base casts we recover the retraction property as well. In this setting, the behavior of casts is fully characterized by their relation to the (abstractly axiomatized) heterogeneous term precision relation. We then provide a model of our axiomatic system in the style of the previous chapter: we define a homogeneous error approximation relation and then extend it to a heterogeneous relation by the insertion of casts.
- In Chapter 10, we take the heterogeneous term precision as primary: the logical relation is indexed by term precision derivations, showing that we can define *first* a semantic ordering on terms, and then show that the casts satisfy the canonical relationship axiomatized by the lub/glb properties of Chapter 5.

So we have two constructive interpretations of type precision derivations $A \sqsubseteq B$

1. As an embedding-projection pair from A to B.
2. As a *relation* between terms of type A and terms of type B.

These interpretations are closely related.

- Given the embedding-projection pair interpretation of $c : A \sqsubseteq B$, we can define a relation $\sqsubseteq_c$ between A and B by (roughly)

  $$V_A \sqsubseteq_c V_B \quad \text{iff} \quad E_{e,c} V_A \sqsubseteq^{\text{err}} V_B \quad \text{iff} \quad V_A \sqsubseteq^{\text{err}} E_{p,c} V_B$$

  This is the approach taken in Chapter 3.
- Given the relational interpretation $\sqsubseteq_c$ of a precision judgment $c : A \sqsubseteq B$, we have a *specification* for the embedding projection pairs. They should be terms such that the embedding satisfies

  $$E_{e,c} V_A \sqsubseteq^{\text{err}} V_B \quad \text{iff} \quad V_A \sqsubseteq_c V_B$$

  and the projection must satisfy

  $$V_A \sqsubseteq^{\text{err}} E_{p,c} V_B \quad \text{iff} \quad V_A \sqsubseteq_c V_B$$

  This is essentially how GTT (Chapter 5) axiomatizes the behavior of ep pairs, by first axiomatizing the heterogeneous relations $\sqsubseteq_c$ and then axiomatizing the ep pairs with this property. In Chapter 10, we define the logical relation and the embedding-projection pairs by recursion on the derivations and this property is central to the graduality proof.

This relationship between embedding-projection pairs and heterogeneous ordering relations is well known in category theory where the embedding-projection pairs are generalized to *adjunctions* and the relations are generalized to *profunctors*. This might serve as a source of inspiration for generalizations of the GTT approach, perhaps to more general notions of interoperability than gradual typing.

11.3 BLAME

We have not used a notion of *blame* in any of our semantics, despite its prevalence in contract and gradual typing literature. Blame can be seen as a further refinement of gradual typing cast semantics where instead of there being only one, uninformative generic "dynamic type error", the error references a specific piece of code, or specific boundary between two program components, which is "blamed" for the violation. This is intuitively very helpful for tracing errors back to their point of origin, since for function and object types, the error may arise in execution of a completely different portion of code than where the type was written.

Introducing blame into our analysis of casts as embedding-projection pairs does introduce some subtleties into the definition of the error ordering. Throughout this dissertation, we have used a single error, which is the least element of the error ordering. To accommodate for multiple errors, we have two choices.

First, for the purposes of the graduality theorem, we should have an ordering where multiple errors are allowed, but each of them is a "bottom" element of the ordering, and so as a consequence every error is equivalent in the ordering. Since the errors are considered different, this means the ordering is a true preorder rather than a partial order, since it does not satisfy anti-symmetry. This is the correct ordering for stating the graduality theorem though, since the addition of casts can change what component is blamed. Furthermore, this means that we cannot use order-equivalence $\sqsupseteq\sqsubseteq$ in place of observational equivalence, which for convenience we have done throughout the development.

Alternatively, we can have a parameterized ordering $\sqsubseteq^l$ that is an error ordering indexed by a blame label or perhaps a set of blame labels, where the labels in the index are considered to be a least element of the ordering and blame using any other labels are only less than themselves. Then the casts should be parameterized by a blame label as well, and then casts using the label l will form embedding projection pairs with respect to the ordering $\sqsubseteq^l$. This is a refinement over the approach of making all blame labels least elements, and so in particular should imply the graduality theorem using that ordering.

11.4 LIMITATIONS OF THE THEORY

The theory of embedding-projection pairs and error approximation ordering relations explains much of the semantics of gradually typed languages that satisfy the graduality theorem. However, this dissertation is far from complete in explaining the multitude of gradual language designs that arise in practice.

11.4.1 *Subtyping*

One very common feature of gradual languages that we have not explored in this dissertation is *subtyping* and the closely related notions of *union* and *intersection* types, which are used extensively to capture the natural, set-like reasoning that programmers employ when using dynamically typed languages. In particular our intrinsically typed approach to semantics does not lend itself directly to a notion of subtyping. Here I will speculate on what might be a reasonable notion.

First, we argue that the interpretation of a subtyping judgment $A \leq B$ as a coercion should be the same as that of a cast, i.e, to cast from A up to ? and then back down to B.

$$\langle B \twoheadleftarrow ? \rangle \langle ? \twoheadleftarrow A \rangle$$

Intuitively this makes sense because we think of the dynamic type as giving us a universal type that should give a ground truth semantics to all values. We then could define subtying $A \leq B$ to hold when this cast is particularly well-behaved. In particular we certainly do not want a subtyping coercion to ever error. But note that because of higher-order types, it is not sufficient to say that for all values $V_A : A$ that $\langle B \twoheadleftarrow ? \rangle \langle ? \twoheadleftarrow A \rangle V_A$ reduces to some V_B. For instance given any two disjoint types, say $\texttt{Int}$ and $\texttt{Bool} \times \texttt{Bool}$, the cast

$$\langle \texttt{Bool} \to \texttt{Int} \twoheadleftarrow ? \rangle \langle ? \twoheadleftarrow \texttt{Bool} \to (\texttt{Bool} \times \texttt{Bool}) \rangle V$$

will always reduce to a value, but if that value is ever applied to a boolean, the resulting function can never return a value because the output will be of the wrong type.

An alternative would be to say that the cast $\langle B \twoheadleftarrow ? \rangle \langle ? \twoheadleftarrow A \rangle$ must be *hereditarily* pure, it runs to a value without error and when given hereditarily pure inputs it returns hereditarily pure outputs, etc. This definition should clearly be reflexive and transitive, and seems like it should also satisfy the expected rules for function types.

It should then be easy to define the *specification* for unions and intersections of types as least upper bounds and greatest lower bounds with respect to this ordering. However, this will not lead directly to an algorithm for computing unions and intersections of types, or determining if they exist. Instead, this is a correctness condition on a syntactic algorithm.

11.4.2 *Consistency and Abstracting Gradual Typing*

Another common syntactic notion in fine-grained gradually typed languages is *consistency* in its guises of consistent equality and consistent subtyping. These notions have been elegantly explained by the *Abstracting Gradual Typing* framework, based on a Galois connection between gradual types and sets of "static types" in a language. However, this approach is fundamentally based on a static type checker for an untyped language, whereas our semantic approach is based on an intrinsically typed semantics, so it might not be possible to give a satisfactory explanation of these notions of consistency in terms of our semantics. That is to say, perhaps we shouldn't think of consistency as a semantic notion at all, but rather a way to power the "smooth transition" from dynamic to static typing, with the graduality property ensuring that the precise details of where casts are placed is not essential to reasoning about gradually typed programs.

11.5 GENERALITY OF EP PAIRS

We have only detailed a very traditional kind of gradual typing: we have a dynamic type and all types have an embedding-projection pair into it. However one avenue of gradual typing research has been to apply the same language design to other things such as adding effect typing or security typing to a simply typed language. These languages might not have a dynamic type in the traditional sense but instead a "dynamic effect" for example or "dynamic label".

To give some idea of how ep pairs might apply in this case, I'll sketch an approach to dynamic effect typing. Let's say the language features a family of types Eff ϵ A where A is a type of values and ϵ describes effects. For instance $\emptyset$ might be empty meaning no tracked effects, or stdin for reading from an input stream, stdout for printing to an output stream, etc. Then say we have a dynamic effect type $?_\epsilon$ that should correspond to programming in an effectful language.

This can be modeled as follows. Rather than having each type have an ep pair to the dynamic type, each effect type will come equipped with a parametric ep pair between Eff ϵ $A \lhd$ Eff $?_\epsilon$ A. To accommodate the possibility of runtime errors, even the "empty" effect would need to allow for the possibility of runtime errors. Then we would expect similar properties like factorization of embedding-projection pairs etc. Semantically these would likely be modeled as homomorphisms of monads that support a notion of error.

BIBLIOGRAPHY

[1] Daniel Ahman, Neil Ghani, and Gordon D. Plotkin. "Dependent Types and Fibred Computational Effects." In: *Foundations of Software Science and Computation Structures*. 2016. DOI: 10.1007/978-3-662-49630-5_3.

[2] Amal Ahmed. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types." In: *European Symposium on Programming (ESOP)*. Vienna, Austria, Mar. 2006, pp. 69–83.

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. "State-Dependent Representation Independence." In: *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*. Jan. 2009.

[4] Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. "Blame for All." In: *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*. Jan. 2011, pp. 201–214.

[5] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. "Theorems for Free for Free: Parametricity, With and Without Types." In: *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*. 2017.

[6] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. "Reconciling Noninterference and Gradual Typing." In: *IEEE Symposium on Logic in Computer Science (LICS), Saarbruecken, Germany*. 2020.

[7] Jean-Marc Andreoli. "Logic programming with focusing proofs in linear logic." In: *Journal of Logic and Computation* 2.3 (1992), pp. 297–347.

[8] Andrej Bauer and Matija Pretnar. "An Effect System for Algebraic Effects and Handlers." In: *Algebra and Coalgebra in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–16.

[9] Nick Benton. "Embedded Interpreters." In: *Journal of Functional Programming* 15.04 (2005), pp. 503–542.

[10] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript." English. In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Vol. 8586. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 257–281.

[11] John Boyland. "The Problem of Structural Type Tests in a Gradually-Typed Language." In: *21st Workshop on Foundations of Object-Oriented Languages*. 2014.

[12] Giuseppe Castagna and Victor Lanvin. "Gradual Typing with Union and Intersection Types." In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 41:1–41:28. ISSN: 2475-1421. DOI: 10.1145/3110285. URL: http://doi.acm.org/10.1145/3110285.

[13] Matteo Cimini and Jeremy G. Siek. "The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. 2016.

[14] Matteo Cimini and Jeremy G. Siek. "Automatically Generating the Dynamic Semantics of Gradually Typed Languages." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. 2017, pp. 789–803.

[15] Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. "Partial Type Equivalences for Verified Dependent Interoperability." In: *International Conference on Functional Programming (ICFP)*. Nara, Japan, 2016.

[16] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. "Foundations of dependent interoperability." In: *Journal of Functional Programming* 28 (2018), e9. DOI: 10.1017/S0956796818000011.

[17] Markus Degen, Peter Thiemann, and Stefan Wehr. "The interaction of contracts and laziness." In: *Higher-Order and Symbolic Computation* 25 (2012), pp. 85–125.

[18] Tim Disney and Cormac Flanagan. "Gradual Information Flow Typing." In: *Workshop on Script-to-Program Evolution (STOP)*. 2011.

[19] Brian Patrick Dunphy. "Parametricity As a Notion of Uniformity in Reflexive Graphs." PhD thesis. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 2002.

[20] Joseph Eremondi, Èric Tanter, and Ronald Garcia. "Approximate Normalization for Dependent Gradual Types." In: *International Conference on Functional Programming (ICFP), Berlin, Germany*. 2019.

[21] Keun-Bang Hou (Favonia), Nick Benton, and Robert Harper. "Correctness of compiling polymorphism to dynamic typing." In: *Journal of Functional Programming* 27 (2017).

[22] Matthias Felleisen. "On the expressive power of programming languages." In: *ESOP'90* (1990).

[23] Luminous Fennell and Peter Thiemann. "Gradual Security Typing with References." In: *CSF*. IEEE Computer Society, 2013, pp. 224–239.

[24] Robby Findler and Matthias Blume. "Contracts as Pairs of Projections." In: *International Symposium on Functional and Logic Programming (FLOPS)*. Apr. 2006.

[25] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions." In: *International Conference on Functional Programming (ICFP)*. Sept. 2002, pp. 48–59.

[26] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. "Semantic Casts: Contracts and Structural Subtyping in a Nominal World." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2004.

[27] Carsten Führmann. "Direct models of the computational lambda-calculus." In: *Electronic Notes in Theoretical Computer Science* 20 (1999), pp. 245–292.

[28] Ronald Garcia and Matteo Cimini. "Principal Type Schemes for Gradual Programs." In: POPL '15. 2015.

[29] Ronald Garcia, Alison M. Clark, and Éric Tanter. "Abstracting Gradual Typing." In: *ACM Symposium on Principles of Programming Languages (POPL)*. 2016.

[30] Jean-Yves Girard. "Locus Solum: From the rules of logic to the logic of rules." In: *Mathematical Structures in Computer Science* 11.3 (2001), 301–506.

[31] Michael Greenberg. "Space-Efficient Manifest Contracts." In: *ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, 2015, pp. 181–194.

[32] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. "Contracts Made Manifest." In: *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*. 2010.

[33] Ben Greenman and Matthias Felleisen. "A Spectrum of Type Soundness and Performance." In: *International Conference on Functional Programming (ICFP), St. Louis, Missouri*. 2018.

[34] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. "Complete Monitors for Gradual Types." In: *PACMPL* 3.OOPSLA (2019), 122:1–122:29. DOI: https://doi.org/10.1145/3360548.

[35] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. "How to evaluate the performance of gradual type systems." In: *Journal of Functional Programming* 29.04 (2005). DOI: 10.1017/S0956796818000217.

[36] Robert Harper. *Dynamic Languages are Static Languages*. 2011. URL: https://existentialtype.wordpress.com/2011/03/19/dynamic-languages-are-static-languages/.

[37] Anders Hejlsberg. *Introducing TypeScript*. Microsoft Channel 9 Blog. 2012.

[38] Fritz Henglein. "Dynamic Typing: Syntax and Proof Theory." In: *Science of Computer Programming* 22.3 (1994), pp. 197–230.

[39] David Herman, Aaron Tomb, and Cormac Flanagan. "Space-efficient gradual typing." In: *Higher-Order and Symbolic Computation* (2010).

[40] Ralf Hinze, Johan Jeuring, and Andres Löh. "Typed Contracts for Functional Programming." In: *International Symposium on Functional and Logic Programming (FLOPS)*. 2006.

[41] Atsushi Igarashi, Peter Thiemann, Vasco Vasconcelos, and Philip Wadler. "Gradual Session Types." In: *International Conference on Functional Programming (ICFP)*. 2017.

[42] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. "On Polymorphic Gradual Typing." In: *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*. 2017.

[43] Lintaro Ina and Atsushi Igarashi. "Gradual typing for generics." In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '11. 2011.

[44] Nico Lehmann and Éric Tanter. "Gradual Refinement Types." In: *ACM Symposium on Principles of Programming Languages (POPL)*. 2017.

[45] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer, 2003.

[46] Paul Blain Levy. "Contextual Isomorphisms." In: *ACM Symposium on Principles of Programming Languages (POPL)*. 2017.

[47] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do Be Do Be Do." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 500–514.

[48] QingMing Ma and John C. Reynolds. "Types, Abstractions, and Parametric Polymorphism, Part 2." In: *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA*. 1991.

[49] Jacob Matthews and Robert Bruce Findler. "Operational Semantics for Multi-Language Programs." In: *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*. 2007.

[50] John C. Mitchell and Gordon D. Plotkin. "Abstract types have existential type." In: *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*. 1985.

[51] Eugenio Moggi. "Notions of computation and monads." In: *Inform. And Computation* 93.1 (1991).

[52] James H. Morris. "Types Are Not Sets." In: *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*. 1973.

[53] Guillaume Munch-Maccagnoni. "Models of a Non-associative Composition." In: *Foundations of Software Science and Computation Structures*. 2014, pp. 396–410.

[54] Hiroshi Nakano. "A modality for recursion." In: *IEEE Symposium on Logic in Computer Science (LICS), Santa Barbara, California*. 2000.

[55] Georg Neis, Derek Dreyer, and Andreas Rossberg. "Non-Parametric Parametricity." In: *International Conference on Functional Programming (ICFP)*. Sept. 2009, pp. 135–148.

[56] Max S. New and Amal Ahmed. "Graduality from Embedding-Projection Pairs." In: *International Conference on Functional Programming (ICFP), St. Louis, Missouri*. 2018.

[57] Max S. New, Dustin Jamner, and Amal Ahmed. "Graduality and Parametricity: Together Again for the First Time." In: *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*. 2020.

[58] Max S. New and Daniel R. Licata. "Call-by-name Gradual Type Theory." In: *FSCD* (2018).

[59] Max S. New and Daniel R. Licata. "Call-by-name Gradual Type Theory." In: *Logical Methods in Computer Science* 16.1 (Jan. 2020).

[60] Max S. New, Daniel R. Licata, and Amal Ahmed. "Gradual Type Theory." In: POPL '19. 2019.

[61] Pierre-Marie Pédrot and Nicolas Tabareau. "The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects." In: *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*. 2020.

[62] Frank Pfenning and Dennis Griffith. "Polarized Substructural Session Types (invited talk)." In: *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 2015.

[63] Gordon Plotkin and Martín Abadi. "A logic for parametric polymorphism." In: *Typed Lambda Calculi and Applications* (1993), pp. 361–375.

[64] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism." In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France*. 1983.

[65] Amr Sabry and Matthias Felleisen. "Reasoning about Programs in Continuation-Passing Style." In: *Conf. on LISP and functional programming, LFP '92*. 1992.

[66] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. "A Theory of Gradual Effect Systems." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden, 2014, pp. 283–295.

[67] Dana Scott. "Continuous lattices." In: *Toposes, algebraic geometry and logic*. 1972, pp. 97–136.

[68] Ilya Sergey and Dave Clarke. "Gradual Ownership Types." In: *ESOP*. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 579–599.

[69] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages." In: *Scheme and Functional Programming Workshop (Scheme)*. Sept. 2006, pp. 81–92.

[70] Jeremy G. Siek and Walid Taha. "Gradual Typing for Objects." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2007.

[71] Jeremy G. Siek and Philip Wadler. "Threesomes, with and Without Blame." In: *ACM Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain: ACM, 2010, pp. 365–376.

[72] Jeremy G. Siek and Philip Wadler. "Threesomes, with and without blame." In: *ACM Symposium on Principles of Programming Languages (POPL)*. 2010, pp. 365–376.

[73] Jeremy Siek, Ronald Garcia, and Walid Taha. "Exploring the Design Space of Higher-Order Casts." In: *European Symposium on Programming (ESOP)*. York, UK: Springer-Verlag, 2009, pp. 17–31.

[74] Jeremy Siek and Sam Tobin-Hochstadt. "The recursive union of some gradual types." In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Springer LNCS)* volume 9600 (2016).

[75] Jeremy Siek, Micahel Vitousek, Matteo Cimini, and John Tang Boyland. "Refined Criteria for Gradual Typing." In: *1st Summit on Advances in Programming Languages*. 2015.

[76] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. "Chaperones and Impersonators: Runtime Support for Reasonable Interposition." In: *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Tucson, Arizona, USA*. 2012.

[77] Eijiro Sumii and Benjamin C. Pierce. "A Bisimulation for Dynamic Sealing." In: *ACM Symposium on Principles of Programming Languages (POPL), Venice, Italy*. 2004.

[78] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. "Is Sound Gradual Typing Dead?" In: *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*. 2016.

[79] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. "Gradual typing for first-class classes." In: ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). 2012.

[80] Satish Thatte. "Quasi-static typing." In: *ACM Symposium on Principles of Programming Languages (POPL)*. 1990, pp. 367–381.

[81] Sam Tobin-Hochstadt and Matthias Felleisen. "Interlanguage Migration: From Scripts to Programs." In: *Dynamic Languages Symposium (DLS)*. Oct. 2006, pp. 964–974.

[82] Sam Tobin-Hochstadt and Matthias Felleisen. "The Design and Implementation of Typed Scheme." In: *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*. 2008.

[83] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. *Typed Racket Reference*. Accessed: 2019-10-30. URL: https://docs.racket-lang.org/ts-reference/Typed_Units.html.

[84] Matías Toro, Ronald Garcia, and Éric Tanter. "Type-Driven Gradual Security with References." In: *ACM Transactions on Programming Languages and Systems* 40.4 (Dec. 2018). URL: http://doi.acm.org/10.1145/3229061.

[85] Matías Toro, Elizabeth Labrada, and Éric Tanter. "Gradual Parametricity, Revisited." In: (2019).

[86] Julien Verlaguet. "Facebook: Analyzing PHP statically." In: *Commercial Users of Functional Programming (CUFP)*. 2013.

[87] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. "Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems." In: *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*. 2017.

[88] Philip Wadler and Robert Bruce Findler. "Well-typed programs can't be blamed." In: *European Symposium on Programming (ESOP)*. York, UK, Mar. 2009, pp. 1–16.

[89] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. "Gradual Typestate." In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. 2011.

[90] Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. "Consistent Subtyping for All." In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 3–30. ISBN: 978-3-319-89884-1.

[91] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. "Static Contract Checking for Haskell." In: *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*. 2009.

[92] Noam Zeilberger. "The Logical Basis of Evaluation Order and Pattern-Matching." PhD thesis. Carnegie Mellon University, 2009.

DECLARATION

Put your declaration here.

Boston, 2020

Max Stewart New

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of April 22, 2021 (classicthesis v4.6).