Notions of Stack-manipulating Computation and Relative Monads

YUCHEN JIANG, University of Michigan, USA
RUNZE XUE, University of Michigan, USA
MAX S. NEW, University of Michigan, USA

We present *relative monads* in a call-by-push-value (CBPV) calculus as a common abstraction for stack-based implementations of effects. This brings the powerful abstraction of monadic programming to lower-level stack-based languages. We demonstrate the applicability of the relative monad by modeling stack-based implementations of common monads used in functional programming as relative monads such as exceptions, state, continuations and free monads. All examples are executable in Zydeco, a stack-based functional language we have designed and implemented based on CBPV.

We also show that relative monads in CBPV are more compositional than monads in pure languages, since they are relative to an ambient notion of effect. We show that any program written in CBPV can be translated to one that uses an arbitrary relative monad, showing that the entire language of CBPV can be "overloaded" and interpreted using an arbitrary effect. One consequence is that contrary to the situation with monads in pure languages, all relative monads in CBPV can be mechanically extended to a relative monad transformer. We outline potential applications to functional programming and verified compilation using stack-based intermediate representations.

1 INTRODUCTION

Since Moggi's seminal work [Moggi 1991], monads have become a wildly successful tool in two main areas of programming languages. First, as Moggi originally showed, denotational models of functional programming languages with effects are naturally modeled using monads. Secondly, monads have become an indispensable programming abstraction for effects in functional languages, most notably in Haskell, where core I/O primitives are made available through a monadic interface [Peyton Jones and Wadler 1993; Wadler 1990].

The power of monads comes from their ability to "overload the semicolon" by instantiating a quite simple structure: a type constructor, return and bind, satisfying some natural equations. For programmers this means re-using do notation and combinators across many different instantiations of the monad interface. For semanticists this means providing a simple recipe to construct mathematical models of effectful languages. These are really two views on the same idea: the monadic programmer is working with a shallowly embedded effectful programming language. Additionally, programmers and semanticists alike have developed further refinements of monads such as monad transformers and algebraic effects to work with programs or semantics that mix multiple effects in a compositional way while still providing sensible equational reasoning principles that programmers intuitively understand [Liang et al. 1995; Plotkin and Power 2001].

But the purview of effectful programming is hardly limited to abstract mathematical semantics or embedded monadic programming. Most popular programming languages allow for relatively free use of several effects: local and global mutable state, exceptions, threads, and operating system services to name a few. The implementation of these effects is handled not by a source-language programmer implementing a high-level monad interface but instead by the language implementor.

Authors' addresses: Yuchen Jiang, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, lighght@umich.edu; Runze Xue, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, cactus@umich.edu; Max S. New, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, maxsnew@umich.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

While some effects may be directly implemented through dedicated hardware mechanisms such as interrupts, most effects are essentially virtualized, not unlike those of the embedded monadic programmer. A major difference is that the language implementor typically has a much lower-level view of this virtualization. Effects are implemented using various techniques for *stack manipulation*: stack walking, stack swapping, or passing of multiple continuations. Designing these effect implementations thus necessitates thinking at a lower level of abstraction than allowed in high-level languages, which by design limit the control that programmers have over the stack structure. However, some connections between monads and stacks are understood: correspondences between monadic evaluators and stack-based abstract machines [Ager et al. 2005], as well as efficient monad implementations in Haskell [Kiselyov and Ishii 2015].

We posit that adapting the monad interface to directly apply to lower-level languages such as stack-based intermediate representations, could allow for the same modularity and compositionality benefits of monads to be brought to bear on verified language implementation and low-level programming. For instance, common intermediate representations could be instantiated with different monad implementations to get backends supporting different language features. Further, monad algebras and monad homomorphisms could be used to allow for programs written in languages with different effects to safely call each other [Patterson et al. 2023].

In this paper, we show that it is possible to bring the benefits of the monad abstraction to lower-level stack-based implementations of effects. We approach the problem by working in a calculus for stack-manipulation: a polymorphic variant of Levy's call-by-push-value (CBPV). Just as λ calculus is the "standard model" of pure functional programming, we argue that CBPV should be seen as a standard model for stack-manipulating computations. The key is that CBPV has two kinds of types: the value types, which classify first class data that can be passed and returned in functions, and the computation types, which classify programs that manipulate machine states. In this work we take a *dual* view of computation types and view them as *stack types* that classify stack structures with which a computation interacts[1].

We demonstrate stack-based programming abstractions and examples using a language we are developing based on CBPV called Zydeco. Zydeco extends CBPV with nominal data and codata types as well as $\forall/\exists$ polymorphism in the style of System F_ω. We have developed a basic language implementation including a bidirectional type checker, basic I/O primitives and a stack-based interpreter, and all of our code examples are executable Zydeco code. We describe the syntax and a stack-machine operational semantics for Zydeco in Section 3 to demonstrate how CBPV programs can be viewed in terms of stack manipulation.

We give example implementations of effects, adapting techniques from monadic programming and language implementation. We then observe that the monadic interface is not directly applicable as an abstraction because the notion of "monad" we use has the wrong kind. In Zydeco, the natural notion of "monad" has kind $\mathsf{VType} \to \mathsf{CType}$, as it takes a value type A to a computation type that describes how the stack must be structured for an A-returning computation to run. That is, our "monads" are not even endofunctors. Fortunately, as Altenkirch, Chapman and Uustalu showed, "monads need not be endofunctors", in that we can generalize monads to *relative* monads [Altenkirch et al. 2010]. We give a definition of a relative monad in Zydeco and show some common programming patterns: several common monads, algebras of a monad and monad transformers can all be adapted to this relative setting.

We then demonstrate how the power of monadic abstraction can be adapted to our relative monads in CBPV. First, we show that relative monads provide an *even stronger* form of overloadability than ordinary monads: *all constructions on computation types* can be overloaded to an

[1] this duality can be viewed as a variant of linear logic duality, see Egger et al. [2014]

arbitrary relative monad. That is, for any relative monad, there is a CBPV to CBPV translation that reinterprets the ambient notion of effect in the source as the given relative monad[2].

A consequence of this perfect overloadability of CBPV is that unlike the situation in λ-calculus, *any* relative monad in CBPV can be mechanically extended to a relative monad transformer. We show that this has application to monad transformers for pure languages. Since pure languages can be seen as a trivial model of CBPV with no effects, any relative monad (transformer) in CBPV can be interpreted as a monad (transformer) in λ-calculus, and so any relative monad in CBPV determines a monad transformer in λ-calculus. This gives an intuitive explanation for how standard monad transformers arise in practice.

The remainder of the paper is structured as follows:

(1) In Section 2 we discuss related work.
(2) In Section 3 we describe Zydeco, our concrete syntax for CBPV programming and introduce CBPV programming features by example before providing formal syntax and operational semantics as a stack machine.
(3) In Section 4 we introduce relative monads as a programming abstraction in Zydeco and demonstrate through several examples how we can use Zydeco's stack-machine semantics to reason about different implementations of stacks for effectful computations.
(4) In Section 5 we introduce the abstractions of *algebras*, which give a method for constructing stacks for effects and *relative monad transformers* which give ways to combine monads together.
(5) In Section 6 we prove a theorem we call the *fundamental theorem of CBPV relative monads* which says that all constructs in CBPV can be overloaded to an arbitrary monad.
(6) In Section 7, by applying the fundamental theorem we show that all relative monads in CBPV can be extended to relative monad transformers.
(7) In Section 8 we discuss future work.

2 RELATED WORK

Our Zydeco language and semantics is based on Levy's original CBPV syntax and stack-machine semantics [Levy 2001]. Downen and Ariola [Downen and Ariola 2018] further discussed how different calling conventions can be modeled as conservative extensions to Levy's CBPV through the shifts between positive and negative types, which is similar to our use of CBPV for encoding stack representations. Binder et al. [Binder et al. 2022] introduced the concept of data and codata symmetry and transformations back and forth between the two, named defunctionalization and refunctionalization. The syntax of codata declaration and the stack semantics in Zydeco are inspired by their work.

The notion of relative monad we use in this work is relative to a profunctor $J : \mathcal{V}^{op} \times \mathcal{E} \to Set$, which is not the most well-studied from the literature where J is instead assumed to be a *functor* $J : \mathcal{V} \to \mathcal{E}$ [Altenkirch et al. 2010; Arkor and McDermott 2023a]. If we assume the CBPV model we work with has all returner types, then our notion of relative monad is equivalent to a monad relative to F. However, the formulation in terms of a profunctor more directly corresponds to the natural syntactic formulation we use and demonstrates that the concept of a CBPV relative monad makes sense even if the F type is not assumed to exist. The formulation in terms of a profunctor has been noted by Levy in a talk, and Arkor and McDermott point out that it is equivalent to a monad $\hat{T} : \mathcal{V}^{op} \to Set^{\mathcal{E}}$ relative to $J : \mathcal{V}^{op} \to Set^{\mathcal{E}}$ that happens to be representable in that T factors through the the Yoneda embedding: $\hat{T} = Y \circ T^{op}$ where $T : \mathcal{V} \to \mathcal{E}$ [Levy 2019].

[2]We have not yet implemented this translation for Zydeco as it requires advanced features like a module system or typeclasses features that have not been implemented. We discuss plans for future implementation in Section 8.1

Free monads and their efficient implementations are covered in this paper. Same as us, the key idea behind freer monad optimization is defunctionalization. By delaying the evaluation of continuations (or the first argument of fmap), no fmap is need in the process of freer monad construction, thus improving the overall performance. But different from the traditional tree-structured free monad and its optimization under the name *extensible effects* [Kiselyov and Ishii 2015], our solution utilizes the stack from CBPV, which is a heterogeneous list.

3 ZYDECO: A CBPV LANGUAGE FOR STACK MANIPULATION

In this section, we introduce Zydeco, a stack-based functional language based on CBPV. We introduce the concrete syntax of Zydeco by a few examples, and then discuss its abstract syntax and operational semantics using a stack machine.

3.1 Zydeco by Example

Zydeco is a ML/Haskell-like programming language but based on call-by-push-value (CBPV) [Levy 2001] as its core calculus, rather than call-by-value or call-by-name/need evaluation. CBPV introduces a distinction between *values* and *computations* which have their own corresponding notions of *value types* and *computation types*. This separation allows for call-by-value and call-by-name evaluation to be encoded using a mix of the two. Values are inert data that can be passed around as first-class objects, and value types classify the possible forms this data may take. In Zydeco, the value types, typically written as A, A' include both the usual basic data types like integer, string, but also user-defined algebraic data types (sums of products) and the thunk type Thunk B, whose values suspended computations, i.e., closures, of computation type B.

Computations are "imperative" programs that manipulate machine states. In Zydeco, we think of these machine states as given by the stack in the stack machine semantics, and the computation *types* classify what the computation is allowed to do to the machine state, or viewed in the dual, the structure that the stack is allowed to take. First, we have a base type called OS that is the type of computations that interact with the operating system. Next, there is the return type Ret A which classifies computations which may return A values. We think of the stacks of Ret A to be opaque *continuations*.

Lastly, computation types include user-defined *codata* types, which are a dual construct to the familiar algebraic data types[Abel et al. 2013; Binder et al. 2022]. Codata types allow for a kind of "user defined stack frames". Dual to how data types values are introduced using *constructors* and eliminated by pattern matching on the constructor, codata types are eliminated by *destructors* are introduced by *copattern-matching*, a form of pattern-match on the structure of the stack. One can think of codata as a generalized representation of (polymorphic) function types that includes a notion of dispatching. Section 3.1 illustrates encodings of different variations of function types[3]. First, the ordinary CBPV function type A -> B is a codata type with a single destructor, the function application is to push the destructor .arg onto the stack with the data of the argument to the function. Dually, λ abstraction is a copattern match that pops the .arg off the stack. Next, the optional argument function type A ->? B is encoded as a codata type with two destructors: .some to push the argument onto the stack before running the computation B and .none to run the computation without any argument. To process different destructors in this scenario, a comatch structure copattern-matches the top of the stack, popping off the destructor that it finds. Taking a step further, a variadic function type uses a coinductive codata type. It provides two destructors: .more to push more arguments onto the stack and .done to conclude the function call.

[3]we take some liberties here with the concrete syntax of Zydeco, which as implemented does not allow for user-defined infix operators

Codata types can also be used to implement a form of interfaces as in object-oriented programming. The user can choose from a set of destructors and push one onto the stack to invoke a method, while the implementor pops it off the stack with a copattern match, and "dispatches" the corresponding computation according to what's on the stack.

```
codata A -> B where
  | .arg(A): B
end
```

```
codata A ->? B where
  | .some(A): B
  | .none( ): B
end
```

```
codata A ->* B where
  | .more(A): A ->* B
  | .done( ): B
end
```

Fig. 1. Functions Are Codata Types

Because these types are all codata types, they can be combined together, providing a compositional type language for defining stack structure. For instance we can define a function that takes at least one argument as A ->+ B := A -> A ->* B. As examples program, in Section 3.1, we define two functions. First, abort is a polymorphic function of type forall A. A ->+ Ret A which takes a variable number of arguments and returns the first one. It works by popping the type A and argument x off of the stack, and then defining a loop called unwind that copattern matches to see if there are any remaining arguments. If there are more arguments, it continues, and otherwise it returns the original argument x. Here unwind has type Thunk(A ->* Ret A), i.e., it is a closure and we explicitly execute run it by ! unwind.

The second example is a simple command-line program that prints out a running total of user input numbers to demonstrate basic CBPV programming. Zydeco has a built-in computation type OS which is the type of the main computation to be run, analogous to Haskell's IO (), but notably OS is not a monad. Instead, I/O primitives are given in continuation-passing style. For example, read_int has type Thunk(Thunk(Option Int -> OS) -> Thunk OS), taking a continuation for the int that may be read and executing it. Here rather than using the equivalent Int ->? OS we use Option to demonstrate ordinary pattern matching. To write to the console, we use the write_line function, which has type Thunk(Str -> Thunk(OS) -> OS). Finally, we use a primitive function add of type Thunk(Int -> Int -> Ret Int) for addition. Altogether the code follows a similar pattern to abort: we define a tail recursive loop, using the Int parameter as the state, reading in integers one by one, exiting if the read fails. If we do receive +Some(i), we use a do binding to add it to the old value to get a new state, then print the current state and a message and continue reading. We note here that the primitive CBPV do binding is *not* simply a monadic do. As this example shows, the continuation in the do binding is not of type Return A but instead is the primitive base computation type OS. This is a distinctive feature of CBPV: continuations of a do binding can be *any* computation type. As we will make precise later, this ensures all computation types carry a primitive algebra structure for the relative monad Return.

3.2 Syntax

The abstract syntax of Zydeco is depicted in Figure 3. There are three kinds of contexts, a nominal context Σ of data/codata declarations, a context Δ of type variables with their kinds and a context Γ of term variables with their value types. The notation $(\phi_i)^*$ denotes a sequence of ϕ syntax structures, indexed by a natural number i. A valid Zydeco top-level program is made up of type declarations, term definitions that are values, and a main term of computation. We introduce the kinds, types, values and computations in the type checking and the stacks in the stack machine semantics shortly.

Zydeco has a standard nominal type system, allowing programmers to introduce new types through data and codata declarations. Since the type declarations are mutually recursive, to check all of them, we need to go through all of the type declarations in two passes, namely the extraction

```
def fn abort : forall A. A ->+ Ret A =
  fn A -> fn x -> rec unwind ->
  comatch
  | .more _ -> ! unwind
  | .done -> ret x
  end
end
```

```
main
  (rec (loop: Thunk (Int -> OS)) -> fn sum ->
    ! read_int { fn i? -> match i?
      | +None() -> ! exit 0
      | +Some(i) ->
        do sum <- ! add i sum;
        ! write_int  sum {
        ! write_line " = sum" {
        ! loop sum
        }}
      end })
  0
end
```

Fig. 2. Two Code Examples

$$
\begin{array}{rrcl}
\text{Decl Ctx} & \Sigma & : & \cdot \mid \Sigma,\ G \\
\text{Type Decl} & G & ::= & \text{data } X\ (X_k : K_k)^*\ \text{where}\ (\ \mid C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \text{end} \\
 & & \mid & \text{codata } X\ (X_k : K_k)^*\ \text{where}\ (\ \mid .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^* : B_i\ )^*\ \text{end} \\
 & & & \\
\text{Kind} & K & ::= & \text{VType} \mid \text{CType} \mid (K)^* \to K \\
 & & & \\
\text{Type Ctx} & \Delta & ::= & \cdot \mid \Delta,\ X : K \\
\text{Type Var} & X & : & \text{Id} \\
\text{Type} & T, A, B & ::= & X\ (T_i)^* \mid \text{Thunk } T \mid \text{Int} \mid \text{String} \mid \text{Ret } T \mid \text{OS} \\
 & & & \\
\text{Term Ctx} & \Gamma & ::= & \cdot \mid \Gamma,\ x : T \\
\text{Term Var} & x & : & \text{Id} \\
\text{Value} & V & ::= & x \mid \{\ M\ \} \mid C\ (T_i)^*\ (V_i')^* \mid \text{halt} \\
\text{Computation} & M & ::= & \text{ret } V \mid \text{do } x \leftarrow M\ ;\ M \mid !\ V \\
 & & \mid & \text{let } x = V \text{ in } M \mid \text{rec } f.\ M \\
 & & \mid & \text{match } V\ (\ \mid C_i\ (X_{ij})^*\ (x_{ij'})^* \to M_i\ )^*\ \text{end} \\
 & & \mid & \text{comatch}\ (\ \mid .D_i\ (X_{ij})^*\ (x_{ij'})^* \to M_i\ )^*\ \text{end} \\
 & & \mid & M\ .D\ (T_i)^*\ (V_i')^* \\
\text{Stack} & \mathsf{S} & ::= & \bullet \mid \text{Kont}(x\ .\ M) :: \mathsf{S} \mid \text{Dtor}(.D\ (T_i)^*\ (V_i')^*) :: \mathsf{S}
\end{array}
$$

Fig. 3. The Syntax of Zydeco

pass and validation pass. The extraction pass collects kinding information from the type declaration context Σ into a type context Δ, while the validation pass verifies the correctness of type usages, ensuring all types are well-kinded as per the kinding judgments detailed in the following section. The rules fairly standard and found in the appendix Appendix A.

While the actual implementation of the Zydeco type system is based on a bidirectional type system [Dunfield and Krishnaswami 2022], for simplicity and readability, a simplified version of the type system is presented. In Zydeco, all types can be classified by kind - VType for value types and CType for computation types. For clarity, when a type is distinctly a value type, we denote it with A to signify our intention, and implicitly add a premise to check is VType kinded. Similarly, for computation types, B is used under the assumption of it being CType for analogous reasons. We note that we depart slightly from Levy's original CBPV syntax to emphasize our operational view of CBPV. Levy's original notation is briefer, writing Ret A as FA and Thunk B as UB. There are two base value types Int and String, and one base computation type OS.

We present the typing rules in Figure 4, which are mostly standard for CBPV. First, we have the rules for values, which include variables, thunks, which we denote with "suspenders" following the Frank language [Lindley et al. 2017], and constructors of data types. In practice, we would extend these rules with typing for primitives such as addition/printing. For simplicity, we include just one primitive thunk halt to terminate a program of type OS. Next, we have the rules for computations. The Ret rules are similar to a monad, with ret as the introduction rule and a bind for the elimination, but with an arbitrary computation type allowed for the continuation. We can force a Thunk B to get a B computation. We add standard let and recursion rules, though note that the recursive rule must put a Thunk B into the context, as all variables are of value type. Finally we have rules for pattern matching and copattern matching, which check that the cases all have the same output type and are correctly typed with respect to the type definitions in scope. Lastly the destructor rule applies a destructor to a computation of a codata type.

3.3 Operational Semantics with a Stack Machine

We present the operational semantics of Zydeco in Figure 5, using an abstract stack machine. The configuration for the stack machine is a pair of the computation and the stack. The system halts when the configuration $\langle$! halt $\mid \bullet \rangle$ is encountered. Since we are mainly illustrating the stack structure, we use the substitutions instead of environments for the sake of simplicity. The rules that appear in Figure 5 are standard for a CBPV calculus with a stack machine, similar to Levy's original stack machine semantics. A bind pushes a continuation onto the stack, and dually a return will pop off the continuation and execute it with the provided value. Forcing a thunk executes the suspended computation within. Let, rec and pattern matching are straightforward. Finally, placing a destructor on a computation M .D $(T_i)^*$ $(V_{i'})^*$ pushes the destructor onto the stack and continues executing M, whereas a copattern match correspondingly pattern matches against the current stack to determine which branch to execute.

4 RELATIVE MONADS IN ZYDECO

Next we turn to the question of how to virtualize effectful computations in Zydeco. We see that a pattern similar to ordinary monads arises, but needs to be refined to a *relative monad*. We give an interface for relative monads in Figure 6, described as a computation type with two destructors: ret and bind. This is a straightforward translation of the ordinary monad interface, but crucially, the natural kinding for the monad M is VType -> CType. We describe the appropriate monad laws after a few examples.

4.1 Virtualizing Exceptions

To start, we consider the classic example of an exception monad in order to show how the stack layout of monad implementations can be flexibly declared and optimized. The most straightforward adaptation of the classic exception monad is by returning a value of Either type as shown in Figure 7. Because we are in a CBPV setting, we require the Either to be wrapped in a Ret. Intuitively, Ret represents the computation that passes the Either typed value to the continuation stored on top of the stack. The implementation of this relative exception monad structure is a straightforward translation of the ordinary exception monad structure. The outermost comatch dispatches the call to either the .return "method" or the .bind "method". The part fn .done $\rightarrow$ is a syntactic sugar for comatch with only one destructor.

Although this implementation is common for virtualizing exceptions in Haskell, OCaml and Rust, it has downsides from the perspective of implementing an efficient low-level exception mechanism. The first problem is excessive use of data constructors and pattern matching: every .bind in the execution trace creates a branch upon pattern matching. Since branch misprediction is a major

$\boxed{\Delta \vdash T : K}$ type (constructor) T has kind K under kinding context Δ

$$\frac{\Delta \vdash X : (K_i)^* \to K \qquad \Delta \vdash T_i : K_i}{\Delta \vdash X\ (T_i)^* : K}\ [\textsc{TyTApp}] \qquad \frac{}{\Delta \vdash \mathsf{Int} : \mathsf{VType}}\ [\textsc{TyInt}] \qquad \frac{}{\Delta \vdash \mathsf{String} : \mathsf{VType}}\ [\textsc{TyString}]$$

$$\frac{}{\Delta \vdash \mathsf{OS} : \mathsf{CType}}\ [\textsc{TyOS}] \qquad \frac{\Delta \vdash X : \mathsf{CType}}{\Delta \vdash \mathsf{Thunk}\ X : \mathsf{VType}}\ [\textsc{TyThunk}] \qquad \frac{\Delta \vdash X : \mathsf{VType}}{\Delta \vdash \mathsf{Ret}\ X : \mathsf{CType}}\ [\textsc{TyReturn}]$$

$\boxed{\Delta; \Gamma \vdash V : A}$ V has type A

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ [\textsc{ValueVar}] \qquad \frac{\Delta; \Gamma \vdash M : B}{\Delta; \Gamma \vdash \{\ M\ \} : \mathsf{Thunk}\ T}\ [\textsc{ValueThunk}] \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{halt} : \mathsf{Thunk\ OS}}\ [\textsc{ValueHalt}]$$

$$\frac{\begin{array}{c}\mathsf{data}\ X\ (X_k : K_k)^*\ \mathsf{where}\ (\ |\ C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \mathsf{end} \in \Sigma \\ \exists i, C = C_i \qquad \delta = (T_k/X_k)^* \qquad \delta' = (T_j/X_{ij})^* \qquad \Delta, (X_{ij} : K_{ij})^*; \Gamma \vdash V_{j'} : A_{ij'}[\delta][\delta']\end{array}}{\Delta; \Gamma \vdash C\ (T_j)^*\ (V_{j'})^* : X\ (T_k)^*}\ [\textsc{ValueCtor}]$$

$\boxed{\Delta; \Gamma \vdash M : B}$ M has type B

$$\frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathsf{ret}\ V : \mathsf{Ret}\ A}\ [\textsc{CompuReturn}] \qquad \frac{\Delta; \Gamma \vdash M_1 : \mathsf{Ret}\ A \qquad \Delta; \Gamma,\ x : A \vdash M_2 : B}{\Delta; \Gamma \vdash \mathsf{do}\ x \leftarrow M_1\ ;\ M_2 : B}\ [\textsc{CompuBind}]$$

$$\frac{\Delta; \Gamma \vdash V : \mathsf{Thunk}\ B}{\Delta; \Gamma \vdash !\ V : B}\ [\textsc{CompuForce}] \qquad \frac{\Delta; \Gamma \vdash V : A \qquad \Delta; \Gamma,\ x : A \vdash M : B}{\Delta; \Gamma \vdash \mathsf{let}\ x = V\ \mathsf{in}\ M : B}\ [\textsc{CompuLet}]$$

$$\frac{\Delta; \Gamma,\ x : \mathsf{Thunk}\ B \vdash M : B}{\Delta; \Gamma \vdash \mathsf{rec}\ x.\ M : B}\ [\textsc{CompuRec}]$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash V : X\ (T_k)^* \qquad \mathsf{data}\ X\ (X_k : K_k)^*\ \mathsf{where}\ (\ |\ C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \mathsf{end} \in \Sigma \\ \delta = (T_k/X_k)^* \qquad \forall i.\ \Delta, (X_{ij} : K_{ij})^*; \Gamma, (x_{ij'} : A_{ij'}[\delta])^* \vdash M_i : B\end{array}}{\Delta; \Gamma \vdash \mathsf{match}\ V\ (\ |\ C_i\ (X_{ij})^*\ (x_{ij'})^* \to M_i\ )^*\ \mathsf{end} : B}\ [\textsc{CompuMatch}]$$

$$\frac{\begin{array}{c}\mathsf{codata}\ X\ (X_k : K_k)^*\ \mathsf{where}\ (\ |\ .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^* : B_i\ )^*\ \mathsf{end} \in \Sigma \\ \delta = (T_k/X_k)^* \qquad \forall i.\ \Delta, (X_{ij} : K_{ij})^*; \Gamma, (x_{ij'} : A_{ij'}[\delta])^* \vdash M_i : X\ (T_i[\delta])^*\end{array}}{\Delta; \Gamma \vdash \mathsf{comatch}\ (\ |\ .D_i\ (X_{ij})^*\ (x_{ij'})^* \to M_i\ )^*\ \mathsf{end} : X\ (T_k)^*}\ [\textsc{CompuCoMatch}]$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash M : X\ (T_k)^* \qquad \mathsf{codata}\ X\ (X_k : K_k)^*\ \mathsf{where}\ (\ |\ .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^* : B_i\ )^*\ \mathsf{end} \in \Sigma \\ \exists i, D = D_i \qquad \Delta; \Gamma \vdash T_j : K_{ij} \qquad \delta = (T_k/X_k)^* \qquad \delta' = (T_j/X_{ij})^* \qquad \Delta; \Gamma \vdash V'_j : A_{ij}[\delta][\delta']\end{array}}{\Delta; \Gamma \vdash M\ .D\ (T_j)^*\ (V'_j)^* : B_i[\delta]}\ [\textsc{CompuDtor}]$$

Fig. 4. Kinding and Typing for Zydeco

headache of modern compiler optimization, it's undoubtedly better to avoid branches when we can. The other problem is the redundancy on the +Left(e) case. It's merely reconstructing into the data type for a cold branch, but as a library author, we have no choice but to explicitly state the whole matcher branch while implementing the monad instance.

$\boxed{\langle M \mid \mathrm{S}\rangle \longrightarrow \langle M' \mid \mathrm{S}'\rangle}$ M with stack S steps to M' with stack S′

$$\frac{}{\langle \mathrm{do}\ x \leftarrow M_1\ ;\ M_2 \mid \mathrm{S}\rangle \longrightarrow \langle M_1 \mid \mathrm{Kont}(x\ .\ M_2) :: \mathrm{S}\rangle}\ [\textsc{OpBind}]$$

$$\frac{}{\langle \mathrm{ret}\ V \mid \mathrm{Kont}(x\ .\ M) :: \mathrm{S}\rangle \longrightarrow \langle M[V/x] \mid \mathrm{S}\rangle}\ [\textsc{OpRet}] \qquad \frac{}{\langle\ !\ \{\ M\ \} \mid \mathrm{S}\rangle \longrightarrow \langle M \mid \mathrm{S}\rangle}\ [\textsc{OpForceThunk}]$$

$$\frac{}{\langle \mathrm{let}\ x = V\ \mathrm{in}\ M \mid \mathrm{S}\rangle \longrightarrow \langle M[V/x] \mid \mathrm{S}\rangle}\ [\textsc{OpLet}] \qquad \frac{}{\langle \mathrm{rec}\ x.\ M \mid \mathrm{S}\rangle \longrightarrow \langle M[\{\ \mathrm{rec}\ x.\ M\ \}/x] \mid \mathrm{S}\rangle}\ [\textsc{OpRec}]$$

$$\frac{}{\langle \mathrm{match}\ (C_i\ (T_j)^*\ (V_{j'})^*)\ (\ \mid C_i\ (X_{ij})^*\ (x_{ij'})^* \rightarrow M_i\ )^*\ \mathrm{end} \mid \mathrm{S}\rangle \longrightarrow \langle M_i[(T_j/X_{ij})^*][(V_{j'}/x_{ij'})^*] \mid \mathrm{S}\rangle}\ [\textsc{OpMatch}]$$

$$\frac{}{\langle M\ .D\ (T_i)^*\ (V_{i'})^* \mid \mathrm{S}\rangle \longrightarrow \langle M \mid \mathrm{Dtor}(.D\ (T_i)^*\ (V_{i'})^*) :: \mathrm{S}\rangle}\ [\textsc{OpDtor}]$$

$$\frac{}{\begin{array}{l}\langle \mathrm{comatch}\ (\ \mid .D_i\ (X_{ij})^*\ (x_{ij'})^* \rightarrow M_i\ )^*\ \mathrm{end} \mid \mathrm{Dtor}(.D\ (T_j)^*\ (V_{j'})^*) :: \mathrm{S}\rangle \\ \quad \longrightarrow \langle M_i[(T_j/Xij)^*][(V_{j'}/x_{ij'})^*] \mid \mathrm{S}\rangle\end{array}}\ [\textsc{OpComatch}]$$

Fig. 5. Operational Semantics of Zydeco via a Stack Machine

```
codata Monad (M: VType -> CType) where
  | .return : forall (A: VType) .
    A -> M A
  | .bind : forall (A: VType) (A': VType) .
    Thunk (M A) -> Thunk (A -> M A') -> M A'
end
```

Fig. 6. figure
Definition of the Relative Monad

```
codata Exn (E: VType) (A: VType) where
  | .done : Ret (Either E A)
end
```

```
def fn mexn (E: VType) : Monad (Exn E) =
  comatch
  | .return A a -> fn .done -> ret +Right(a)
  | .bind A A' m f ->
    do a? <- ! m .done;
    match a?
    | +Left(e) -> fn .done -> ret +Left(e)
    | +Right(a) -> fn .done -> ! f a
    end
  end
end
```

Fig. 7. Exception Monad using Sum Types

One well-known optimization is for a potentially erroring computation to be implemented by passing two continuations: the ordinary return continuation and the exception continuation. This technique is sometimes called "double-barreled" continuation-passing style [Thielecke 2001]. We prefer to refer to it as *Church-encoding*. We can easily tell from the implementation of the monad interface that different from the last solution, no intermediate data constructor is produced, eliminating unnecessary conditional branching as the control flow is carried out by invoking the correct continuation directly. As we are in a continuation-passing style now, the stack grows only by the continuations themselves growing bigger and bigger along the way, eventually being run when a return or raise is executed.

A third method for implementing raising of exceptions is via explicit stack-walking to find the nearest enclosing exception handler. We can encode this structure as well in Zydeco by using a coinductive codata type, which we can view as "defunctionalizing" the continuations into explicit stack

```
codata ExnK (E: VType) (A: VType) where
  | .run : forall (R: CType) .
    Thunk (E -> R) -> Thunk (A -> R) -> R
end

def fn mexnk (E: VType) : Monad (ExnK E) =
  comatch
  | .return A a ->
    fn .run R ke ka ->
    ! ka a
  | .bind A A' m f ->
    fn .run R ke ka ->
    ! m .run R ke
    { fn a -> ! f a .run @(R) ke ka }
  end
end
```

Fig. 8. Exception Monad using Church Encoding

frames [Reynolds 1972; Wand 1980]. Figure 9 shows the final version that describes continuations as different handlers. By removing the unnecessary pair of handlers in favor of dedicated handlers, bind is simply pushing .kont onto the stack, but return and fail inspect and traverse the stack of handlers, and apply them accordingly. The user push .try handlers onto the stack to perform error handling.

```
codata ExnDe (E: VType) (A: VType) where
  | .try : forall (E': VType) .
    Thunk (E -> ExnDe E' A) -> ExnDe E' A
  | .kont : forall (A': VType) .
    Thunk (A -> ExnDe E A') -> ExnDe E A'
  | .done : Ret (Either E A)
end

def rec fn mexnde (E: VType) : Monad (ExnDe E) =
  comatch
  | .return -> fn A a ->
    comatch
    | .try -> fn E' k -> ! mexnde @(E') .return @(A) a
    | .kont -> fn A' k -> ! k a
    | .done -> ret +Right(a)
    end
  | .bind -> fn A A' m f ->
    ! m .kont @(A') f
  end
end
```

Fig. 9. Exception Monad, defunctionalized

4.2 Relative Monad Laws

Though we've shown the type signature of the relative monad in Zydeco and seen it in practice in the implementation of the "exception monad" in Zydeco, we haven't yet stated the equations that need to be held. To define a monad, one needs to define its Kleisli triple and prove that the monad laws hold. We've shown three instances of Kleisli triples for the relative monad in Zydeco, giving the type constructor M (the object mapping) of kind VType $\to$ CType, the return function *return* (the *unit*) of type $A \to M\ A$ for any A : VType, and the bind function *bind* (the *Kleisli extension*) of type Thunk $(M\ A) \to$ Thunk $(A \to M\ A') \to M\ A'$ for any A, A' : VType. The codata type declaration of Monad in Figure 6 captures the interface of the Kleisli triple by describing its stack layout. But implementing an instance of Monad that's not enough to show that it's a relative monad. The relative monad laws must also hold.

Definition 4.1. An implementation of return and bind are *lawful* if they satisfy the following equational principles:

(1) **The right unital law:** ! bind { ! return a } $f \equiv$! $f\ a$ for any $a : A$ and f : Thunk $(A \to M\ A)$,
(2) **The left unital law:** ! bind m return $\equiv$! m for any m : Thunk $M\ A$,
(3) **The associativity law:** ! bind { ! bind $m\ f$ } $g \equiv$! bind m { fn x $\to$! bind { ! $f\ x$ } g } for any f : Thunk $(A \to M\ A')$, g : Thunk $(A' \to M\ A'')$, and m : Thunk $(M\ A)$,
(4) **The linear bind law:** do $m \leftarrow$! tm ; ! bind $m \equiv$! bind { do $m \leftarrow$! tm ; ! m } for any tm : Thunk (Ret (Thunk $(T\ A)$)).

The first three laws are the same as the laws for ordinary monads, and correspond to intuitive program equivalences for effectful programming. The third, the linear bind law, is a new law that is necessary in a system like CBPV which may have ambient effects, as opposed to ordinary monads which are defined in a "pure" language. Intuitively, bind being linear states that it uses its first input *strictly* and never again. For instance, if bind simply pushes something onto the stack and executes the thunk, then it will be linear. The precise formulation used here says that for tm a "double thunk", it doesn't matter if we evaluate tm first, and bind the resulting thunk or if we bind a thunk that will evaluate tm each time it is called. This formulation is taken from prior work on CBPV [Levy 2017; Munch-Maccagnoni 2014]. One motivation for why the linearity rule is necessary is that we want a relative monad to "overload" not just the do notation that the primitive Ret type provides, we also want all of the *reasoning principles* that the Ret provides to hold for relative monads. And the bind for Ret simply pushes a continuation onto the stack and executes the input, so it is linear.

We can verify that the first two exception monads we provided satisfy the monad laws using CBPV $\beta\eta$ equality or parametric reasoning, but the defunctionalized exception monad implementation does not! The reason is that by defunctionalizing the stack of continuations, we expose low-level details and so "non-standard" computations can inspect the stack and observe, for instance, the number of frames pushed onto the stack. As a result, while the right unital law and bind linearity law hold, the left unital law and the associativity law fail, with a counter-example provided in Figure 10. This counter example counter-example works similarly to abort, looping through and counting the number of stack frames using an accumulator. To show observational difference, we can pass to bench two functions that should be equal under the left unital law or the associativity law, and look at the program output. '! count-kont', as its name indicates, performs a stack walk and distinguishes impls by their number of .kont frames.

```
def rec fn count_kont
  (E: VType) (A: VType) (i: Int)
  (fi: Thunk (Int -> ExnDe E A))
: ExnDe E A =
  comatch
  | .try -> fn E' ke ->
    ! count_kont @(E') @(A) i
    { fn i -> ! fi i .try @(E') ke }
  | .kont -> fn A' ka ->
    do i' <- ! add i 1;
    ! count_kont @(E) @(A') i'
    { fn i -> ! fi i .kont @(A') ka }
  | .done ->
    ! fi i .done
  end
end
```

```
alias ExnCounter = ExnDe Unit Int end

def fn bench
  (impl: Thunk (Thunk (ExnCounter) -> ExnCounter))
: Ret Int =
  do i? <- ! impl {
    ! count_kont @(Unit) @(Int) 0
    { ! mexnde @(Unit) .return @(Int) }
  } .done;
  match i?
  | +Left(e) -> ret -1
  | +Right(i) -> ret i
  end
end
```

Fig. 10. Counter example counter-example

The root cause of the problem with count_kont is that it involves unrestricted manipulations on the stack frame, yet a canonical instance of the exception monad, when executed, should only compute its result -either an exception or a valid value-and react to one of the destructors correspondingly. We henceforth propose a canonicity condition for these instances as shown below, where M is an arbitrary computation of type Exn E A.

$$
\begin{aligned}
M \equiv\ & \mathsf{do}\ x \leftarrow M\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid +\mathit{Left}(e) \longrightarrow !\ \mathsf{fail}\ e \\
& \mid +\mathit{Right}(a) \longrightarrow !\ \mathsf{return}\ a
\end{aligned}
$$

We verify in Appendix B that this condition ensures the monad laws hold. This is of course a strong restriction, and essentially ensures the implementor of a Exn E A computation only interacts with the monad by calling fail, return, and bind,

4.3 More Examples of Relative Monads in Zydeco

Next we demonstrate how to adapt several classical monads to relative monads in Zydeco. Here we list the type signatures of the monads that one can implement in Zydeco. We omit most of the implementations of the monads due to their similarity to the examples of exception monad shown, but they are included in the supplementary material.

First, the primitive return type in Zydeco Ret is itself a relative monad. The implementation is shown in Figure 11. The return of a is simply ret a, and the bind utilizes the primitive notion of do-binding. In fact, the Ret monad plays the same role in Zydeco that the Identity monad does in Haskell, representing "no effects" or at least no more than the ambient notion of effect allowed by our primitives.

```
def fn mret : Monad Ret =
  comatch
  | .return A a -> ret a
  | .bind A A' m f ->
    do a <- ! m;
    ! f a
  end
end
```

Fig. 11. The Implementation of Ret Monad in Zydeco

We give the types of the two continuation monads Kont and PolyKont and state monad State in Figure 12. The first continuation monad Kont is the one with fixed answer type R. The intuition is that a stack for Kont is an R stack with a continuation on top. Since R is fixed, the computation can interact with the stack before invoking the continuation (if ever). For instance, by picking R to be OS, we can get Zydeco's analog of Haskell's IO monad. The second continuation monad PolyKont looks similar at first to Kont but the R parameter is universally quantified. Since the R is universally quantified, by parametricity the computation cannot interact with the R stack *except* by calling the continuation. For this reason, using parametric reasoning it can be proven that PolyKont is equivalent to Ret, and so we might call PolyKont the church-encoded Ret type [Møgelberg and Simpson 2009; Reynolds 1983]. For this reason it is also possible to leave the Ret type out of Zydeco as a primitive type entirely, since it can be defined as this Church encoding. Finally, the state monad provides a "thread-local" state that can be read and mutated by the computation. Reading off of the stack semantics, the state is conceptually updated "in-place".

```
codata Kont (R: CType) (A: VType) where
  | .run : Thunk (A -> R) -> R
end

codata PolyKont (A : VType) where
  | .bind : forall R. Thunk (A -> R) -> R
end

codata State (S: VType) (A: VType) where
  | .run : forall (R: CType) .
    Thunk (A -> S -> R) -> S -> R
end
```

Fig. 12. The Types of State Monad and Continuation Monads in Zydeco

Next, we turn to another general class of monads, *free monads* generated by a collection of operations. A free monad records a sequence of computations with giving them any a priori

```
alias KRead (R: CType) =
  Thunk (Unit -> Kont R String)
end
alias KWrite (R: CType) =
  Thunk (String -> Kont R Unit)
end
alias KFail (E: VType) (R: CType) =
  Thunk (E -> Kont R Empty)
end
codata Free (A: VType) where
  | .run : forall (R: CType) .
    KRead R -> KWrite R -> KFail String R -> Kont R A
end

def fn mfree : Monad Free' =
  comatch
  | .return A a ->
    fn .run R read write fail ->
    fn .run k -> ! k a
  | .bind A A' m f ->
    fn .run R read write fail ->
    fn .run k' ->
    ! m .run @(R) read write fail
    .run {
      fn a ->
      ! f a .run @(R) read write fail
      .run k'
    }
  end
end
```

Fig. 13. An Example of The Free Monad in Zydeco

```
def fn read (s: Unit) : Free String =
  fn .run R read write fail -> ! read +Unit()
end

def fn write (s: String) : Free Unit =
  fn .run R read write fail -> ! write s
end

def fn fail (s: String) : Free Empty =
  fn .run R read write fail -> ! fail s
end

def fn read_os (_: Unit) : Kont OS String =
  fun .run k -> ! read_line k
end

def fn write_os (s: String) : Kont OS Unit =
  fun .run k -> ! write_line s { ! k +Unit() }
end

def fn fail_os (s: String) : Kont OS Empty =
  fun .run k -> ! panic s
end
```

Fig. 14. Operations and Handlers for a Free Monad

semantic meaning. It allows the user to declare the computations in mind and later provide implementations of handlers for these computations. Different from the traditional tree-structured free monad like in Haskell, the free monad in Zydeco consists of a function from a tuple of handlers on the stack to a continuation monad at the bottom. Intuitively, different handlers can be passed at the "runtime" of the computation, providing different behavior, context, or in general, semantics, for the recipe of monadic computations that the user declared in advance; while the continuation at the end is both the carrier of the monadic operations provided by the user and a chance of interaction with the side effects brought by the handlers. The constraint on the handlers is that the handlers have the same base stack as the continuation.

For example, as shown in Figure 13, there are three handlers on the stack of Free, each responsible for handling read, print, and fail effects correspondingly. In an impure language, read would have type Unit $\to$ String. Translated under the CPS setup, we get Unit $\to$ (String $\to$ R) $\to$ R. Further adopting it into CBPV leads us to its form presented in the example. Similar translation happens to print and fail. fail also demonstrates how to encode an effect that can't return a value. To use the free monad, we also need to define both how to construct the free monad and how to interpret it, which is shown in Figure 14.

```
codata Algebra (M: VType -> CType) (R: CType) where
  | .bindA : forall (A: VType) .
    Thunk (M A) -> Thunk (A -> R) -> R
end
```

Fig. 15. Algebra of a Relative Monad

```
def fn alg_ret (R: CType): Algebra Ret R =
  comatch
  | .bindA A m k ->
    do a <- ! m; ! k a
  end
end
```

```
def fn alg_mo
  (M: VType -> CType)
  (V: VType)
  (mo: Thunk (Monad M))
: Algebra M (M V) =
  comatch
  | .bindA A m k ->
    ! mo .bind @(A) @(V) m k
  end
end
```

Fig. 16. Two "trivial" algebra structures

5 ALGEBRAS OF RELATIVE MONADS AS THE HANDLERS FOR STACK-BASED EFFECTS

Relative monads allow the programmer to express stack-manipulating computations, as demonstrated in the previous section. Nevertheless, there are times when we desire to compute the monad against a different stack provided by the user. Such need is fulfilled by algebras of relative monads, whose definition is given in Figure 15. Intuitively, an algebra is a version of bind that works with an output type not necessarily of the form M A. In terms of stacks, an algebra for M with carrier type R gives a method for taking an R stack and a continuation U(A -> R) and constructing a stack for an M A. Since algebras generalize bind, they should also generalize the monad laws for bind. Specifically, all but the left unital law make sense and should be satisfied.

Definition 5.1. An algebra is *lawful* if it satisfies the following equational principles:

(1) **The right unital law:** ! bindA { ! return a } $f \equiv$! f a for any $a : A$ and f : Thunk $(A \to M\,B)$,
(2) **The associativity law:** ! bindA { ! bind m f } $g \equiv$! bindA m { fn x $\to$! bindA { ! f x } g } for any f : Thunk $(A \to M\,A')$, g : Thunk $(A' \to B)$, and m : Thunk $(M\,A)$,
(3) **The linear bind law:** do $m \leftarrow$! tm ; ! bind $m \equiv$! bind { do $m \leftarrow$! tm ; ! m } for any tm : Thunk (Ret (Thunk $(T\,A)$)).

In Figure 16, we give two of the simplest algebras of relative monads. First, we can define the algebra of the Retmonad for any computation R using the built-in do notation. And second, every monad in Zydeco has an algebra to itself, using its own bind to implement bindA.

In fact, we can extend algebra structures to all codata types. We show representative examples that extend algebra constructions on arrow ($\to$) types and with (&) types demonstrated in Figure 17, which covers a wide range of computation types in Zydeco. By providing the algebra of M : VType $\to$ CType for R : CType, we can define for free the algebra of M : VType $\to$ CType for $A \to R$ for any A : VType. Similarly, we can define the algebra of M : VType $\to$ CType for $L\&R$ for any L, R : CType given both of the algebra on L and R.

We give an example for how to extend algebras to coinductive codata types in Figure 18. Specifically, given an algebra structure on B, we can extend this to an algebra structure on A ->* B by popping the arguments off of the stack and pushing them into the continuation until we finally reach the base case of a B stack, in which case we use B's algebra structure with the accumulated continuation.

What are algebras good for? They allow us to combine non-monadic results with monadic computations. In fact we already have seen an example of this practice with the built-in monad: our

```
def fn alg_arrow
  (M: VType -> CType) (R: CType) (A: VType)
  (alg: Algebra M R)
: Algebra M (A -> R) =
  comatch
  | .bindA A' m f -> fn a ->
    ! alg .bindA @(A) m { fn a' -> ! f a' a }
  end
end

codata With (L: CType) (R: CType) where
  | .l : L
  | .r : R
end
```

```
def fn alg_with
  (M: VType -> CType) (L: CType) (R: CType)
  (algl: Thunk (Algebra M L))
  (algr: Thunk (Algebra M R))
: Algebra M (With L R) =
  comatch
  | .bindA A m f ->
    comatch
    | .l -> ! algl .bindA @(A) m { fn a -> ! f a .l }
    | .r -> ! algr .bindA @(A) m { fn a -> ! f a .r }
    end
  end
end
```

Fig. 17. Algebra structure for -> and &

```
def rec fn alg_var (M: VType -> CType) (R: CType) (alg: Thunk (Algebra M R)) (V: VType)
                   : Algebra M (FnVar V R) =
  comatch
  | .bindA A m ka ->
    let rec fn loop (ka: Thunk (A -> FnVar V R))
    : FnVar V R =
      comatch
      | .more v -> ! loop { fn a -> ! ka a .more v }
      | .done -> ! alg .bindA @(A) m { fn a -> ! ka a .done }
      end
    in ! loop ka
  end
end
```

Fig. 18. Algebra for $A \rightarrow^* B$

```
def fn alg_exn :
  Algebra (ExnDe String) OS
=
  comatch
  | .bindA A m k ->
    do a? <- ! m .done;
    match a?
    | +Left(s) -> ! panic s
    | +Right(a) -> ! k a
    end
  end
end
```

```
main
  (rec (loop: Thunk (Int -> OS)) ->
    fn sum ->
    ! read_int { fn i? -> match i?
      | +None() -> ! exit 0
      | +Some(i) ->
        ! alg_exn .bindA @(Int) { ! add_of i sum } { fn sum ->
        ! write_int sum {
        ! write_line " = sum" {
        ! loop sum }}}
      end }) 0
end
```

Fig. 19. The Echo Sum Server, Extended with a Fallible Addition

echo sum server from Section 3.2 used a do block where the result was OS and not a monad. With algebras of relative monads in place, we can introduce new effects via monads without modifying the structure of the original user program in Zydeco; instead, we can "overload" the do-binding in the original program by providing a new algebra for the new effect. For example, consider what would happen if we changed our addition function to one that raises an exception if the addition overflows, i.e., add_of : Thunk(Int -> Int -> ExnDE Int). In Figure 19 we show that we can rewrite our program from the built-in do-notation to use an explicit bindA operation. In this case the carrier of the algebra is OS, so we need to provide an interpretation of what to do when there is an error. In this case, we simply crash the program by calling a built-in panic function. Replacing the built-in do notation with explicit bindA calls is certainly ugly, but the examples in this section show that in principle we should be able to overload the do entirely. We formalize that this is possible in the next section.

```
codata MonadTrans (T: (VType -> CType) -> VType -> CType) where
  | .monad : forall (M: VType -> CType) .
    Thunk (Monad M) -> Monad (T M)
  | .lift : forall (M: VType -> CType) (A: VType) .
    Thunk (Monad M) -> Thunk (M A) -> T M A
end
```

Fig. 20. The Interface of Monad Transformers in Zydeco

```
codata ExnT (E: VType) (M: VType -> CType) (A: VType) where
  | .done : M (Either E A)
end

def fn mtexn (E: VType) : MonadTrans (ExnT E) =
  comatch
  | .monad M mo ->
    comatch
    | .return A a -> fn .done ->
      ! mo .return @(Either E A) +Right(a)
    | .bind A A' m f -> fn .done ->
      ! mo .bind @(Either E A) @(Either E A') { ! m .done }
      { fn a? ->
          match a?
          | +Left(e) -> ! mo .return @(Either E A') +Left(e)
          | +Right(a) -> ! f a .done
          end }
    end
  | .lift M A mo m -> fn .done ->
    ! mo .bind @(A) @(Either E A) m
    { fn a ->
      ! mo .return @(Either E A) +Right(a) }
  end
end
```

Fig. 21. The Monad Transformer for Ret (*Either E A*)

5.1 Monad Transformers

As a final example of adapting monadic programming to CBPV, we consider monad *transformers*. A monad transformer, defined in Figure 20 is a type $T : (\mathsf{VType} \to \mathsf{CType}) \to \mathsf{VType} \to \mathsf{CType}$ such that for any monad M, $T(M)$ is a monad, and we support a lift function that turns embeds $M\ A$ computations into $T\ M\ A$ computations. So a monad transformer is a way to "add" effects to an input monad M.

Intuitively, all programs in CBPV are defined relative to some ambient base effects, so shouldn't it be the case that any relative monad should also be a monad transformer? Consider a couple of illustrative examples. First, our original exception monad Exn defined in terms of Ret can easily be turned into a monad transformer ExnT in Figure 21 by replacing Ret with the input monad M. Then to implement the monad instance of $T\ M$, change all references of primitive return and do-binding to use corresponding operations inherited from the monad instance of M_T. The lift function can be transformed following the analogy of the left unital law, $!\ \mathsf{bind}\ m\ \mathsf{return} \equiv !\ m$ for any $m : \mathsf{Thunk}\ M\ A$, but use those provided by the monad instance of M, and target $M\ V$ as the output of the transformation.

This transformation is somewhat obvious because it uses Ret explicitly, but what about our Church-encoded monads, which don't explicitly use Ret? The key is that these Church-encoded monads don't mention Ret but they do quantify over computation types. We can mechanically generate a monad transformer by generalizing from quantifying over computation types to quantifying over *algebras of the input monad*. As another example, consider our second exception monad ExnK which we transform into a monad transformer in Figure 22 The instance of monad for $T\ M$ shouldn't be any different from the instance of monad for M_T, and the lift function should be

```
codata ExnKT (E: VType) (R: CType) (M: VType -> CType) (A: VType) where
  | .run : Thunk (Algebra M R) -> Thunk (E -> R) -> Thunk (A -> R) -> R
end

def fn mtexnk (E: VType) (R: CType) : MonadTrans (ExnKT E R) =
  comatch
  | .monad M mo ->
    comatch
    | .return A a -> fn .run alg -> fn ke ka ->
      ! ka a
    | .bind A A' m f -> fn .run alg -> fn ke ka' ->
      do ! m .run alg ke; fn a ->
      ! f a .run alg ke ka'
    end
  | .lift M A mo m -> fn .run alg -> fn ke ka ->
    ! alg .bindA @(A) m ka
  end
end
```

Fig. 22. The Monad Transformer for forall R . Thunk $(E \to R) \to$ Thunk $(A \to R) \to R$ in Zydeco

implemented by a bindA of the algebra of M on R, and passing in the continuation in M_T. Again referring to the previous example of ExnK E A, the monad transformer of it is shown in Figure 22.

6 FUNDAMENTAL THEOREM OF CBPV RELATIVE MONADS

As we have seen, every computation type constructor in Zydeco can be extended to an operation on algebras. This suggests that the built-in do notation in Zydeco could be "overloaded" to support not just the built-in Ret type but an arbitrary relative monad, analogous to the familiar extensibility of do notation for ordinary monads in Haskell. Note that since Zydeco's do notation is based on CBPV this is a more complex overloading. That is, in Haskell, the result type of an expression in do notation is always the monad `m a` and the desugaring uses the monad structure of `m`, but in Zydeco the result type is an arbitrary algebra, and the desugaring of the do notation is determined by the algebra structure.

Because *every* computation type constructor can be extended to algebras, not only should do notation be overloadable, but *all* language constructs (codata type definitions, introduction and elimination rules) should be overloadable in Zydeco to be interpreted relative to an arbitrary relative monad. In this section, we give a partial formalization of this result, that for any relative monad in Zydeco, we can define a syntactic Zydeco to Zydeco translation that reinterprets the ambient effect using the relative monad.

There are two caveats to our formalization which are current obstacles to implementation. First we do not address recursive data/codata types, which we leave to future work to formalize. We do fully expect the translation can be extended to codata types as described in Section 5. Secondly, the translation of polymorphism over computation types (e.g., $\forall$ and $\exists$) is most conveniently expressed to Zydeco containing Σ-kinds, which amounts to upgrading from System F_ω-style polymorphism to an ML-style module system, which we do not currently have implemented in Zydeco [MacQueen 1984; Mitchell and Harper 1988]. We discuss the feasibility of this and other potential implementation strategies such as typeclasses in Section 8.1.

To avoid getting bogged down in irrelevant syntactic details such as name binding and type annotations, we will formalize the overloadability theorem *semantically*. That is, we will define a Zydeco model as an impredicative polymorphic CBPV model with function and Σ kinds, and construct for any Zydeco model with a relative monad, a second model of Zydeco where computation types are modeled as algebras of the relative monad. We call this theorem the *fundamental theorem of CBPV relative monads*.

We assume quite a bit more category theoretic background and familiarity with models of CBPV of the reader in the body of this section and the next than other sections. In particular, to efficiently work with CBPV models, we will extensively use the internal language of presheaf categories to treat an arbitrary CBPV model as if the value types and kinds were given by subcategories of the category of sets without any loss of generality. Readers less familiar with category theory can safely read the beginnings of these sections to get an idea for the main results and skip over the technical details on a first reading. We refer readers to a standard reference on the internal language of presheaf categories such as [Maietti 2005].

6.1 What is a Model of Call-by-push-value?

Next, we introduce the notion of semantic model of CBPV that we will use in our algebra construction. We will do this first by introducing models of the basic judgmental structure (kinds, types, values, stacks and computations) and then characterizing modularly by universal properties when such a model additionally interprets each type constructor. As notation we will use $\mathcal{P}\mathcal{C}$ to mean the category of presheaves $\mathcal{C}^{op} \to Set$. By (contravariant) presheaf we mean a functor $\mathcal{C}^{op} \to Set$ and covariant presheaf on $\mathcal{C}$ we mean $\mathcal{C} \to Set$. If not specified, presheaf means contravariant. We give here four notions of CBPV model each of varying intuitiveness and practical utility:

Definition 6.1. (1) A *unary CBPV model* is a triple $\mathcal{V}, \mathcal{E}, \mathcal{J}$ of a category $\mathcal{V}$, a category $\mathcal{E}$ and a *profunctor* $J : \mathcal{V}^{op} \times \mathcal{E} \to Set$.

(2) A *strong CBPV model* is a triple $\mathcal{V}, \mathcal{E}, C$ of a category $\mathcal{V}$ with finite products, a category $\mathcal{E}$ enriched over $\mathcal{P}\mathcal{V}$, and C an enriched covariant presheaf on $\mathcal{E}$.

(3) A *dependent CBPV model* is a triple $\mathcal{V}, \mathcal{E}, C$ of a natural model of dependent type theory $\mathcal{V} = (\mathcal{V}_c, \mathrm{VTy}, \mathrm{Val})$, a category $\mathcal{E}$ *internal to* $\mathcal{P}\mathcal{V}_c$ and an internal covariant presheaf on $\mathcal{E}$.

(4) A *concrete CBPV model* is a triple $\mathcal{V}, \mathcal{E}, C$ of a universe $\mathcal{V}$ of sets, a category $\mathcal{E}$ and a covariant presheaf $C : \mathcal{E} \to Set$.

(5) A *polymorphic concrete CBPV model* is a concrete CBPV model additionally equipped with a second universe $\mathcal{K}$.

A unary CBPV model is a model of the "unary" fragment of CBPV: that is it models value types and computation types as objects, but it only models restricted judgments of unary values ($x : A \vdash V : A'$), computations $x : A \vdash M : B$ and stacks $\bullet : B \vdash S : B'$. Despite its over-simplicity, the unary model is often the most intuitive: the profunctor J modeling the computation judgment is the central structure of a CBPV model.

Next, a strong CBPV model is a model of simply typed CBPV and is very similar to Levy's original notion used in CBPV adjunction models. The formulation here in terms of presheaf-enrichment is due to Curien et al. [2016]. Strong models generalize unary models because we can reindex $\mathcal{E}$ to be an ordinary category and define $J\,A\,B = C_A(B)$. Dependently typed CBPV models generalize strong CBPV models by allowing value and computation types to be dependent on value types. The value types and pure morphisms are essentially an ordinary model of dependent type theory, and then computation types, computations stacks are internal to presheaves, which intuitively means they are indexed by a context and contain an action of substitution which commutes with stack composition. This notion is related to the one given by Ahman in the same way that comprehension categories are related to natural models in the semantics of dependent type theory[Ahman et al. 2016; Awodey 2018]. It is also quite similar to the theory of dependent CBPV used in the work on Calf, except that the Calf work does not include a stack judgment [Niu et al. 2022].

The final notion, that of a concrete model, breaks the pattern of increasing generality in that at first it looks *much* less general than the prior ones, since it requires the category $\mathcal{V}$ to be a full

subcategory of the category of sets. Despite this, it is the notion of model we will use throughout this paper, because of the following:

Theorem 6.1. *Every dependent CBPV model determines a concrete CBPV model internal to the topos of presheaves on $\mathcal{V}_c$.*

That is, internal to the topos of presheaves, the representable map of presheaves looks like a universe of sets, and an internal category/presheaf look like an ordinary category/presheaf. Since the internal language of a topos can model all of extensional intuitionistic type theory, we freely work in this and the next section in terms of concrete models, knowing that this can be mechanically, if laboriously, translated to corresponding statements about dependent CBPV models. Finally, a polymorphic model additionally adds a universe $\mathcal{K}$ of "kinds", though they do not look much like kinds until we assume some type structure.

Now fix a concrete model $\mathcal{V}, \mathcal{E}, \mathcal{C}$ of CBPV for the remainder of this section. We overload $\mathcal{V}$ to also refer to the full subcategory of *Set* given by sets in the universe $\mathcal{V}$, and we define the profunctor $J\,A\,B = \mathcal{C}(B)^A$. We now give a brief overview of how these constructs correspond to the judgments of CBPV. The intuition we have is that value types T and contexts Γ denote objects of $\mathcal{V}$, i.e., "small" sets, computation types denote objects of $\mathcal{E}$ and computations $\Gamma \vdash M : B$ denote functions $\Gamma \to \mathcal{C}(B)$, i.e., $J\,\Gamma\,B$. The morphisms of $\mathcal{V}$ are "pure functions" between value types, which includes the denotations of syntactic values, and the morphisms of $\mathcal{E}$ are "linear" morphisms between computation types which includes the denotations of syntactic stacks. We write the action of J on morphisms using composition syntax, i.e., we write $(J\,A\,l)(j)$ as $l \circ j$ and $(J\,f\,B)(j)$ as $j \circ f$. This models the composition of the "effectful" computation $j \in J\,A\,B$ with a "pure" morphism f or a linear morphism l, respectively, generalizing the syntactic operations of substituting a syntactic value into a computation or "piling" a stack onto a computation. The functoriality ensures that the implied unit and associativity laws for this notation are satisfied. Finally, kinds k and kind contexts Δ are interpreted as elements of the universe $\mathcal{K}$ and types $\Delta \vdash T : k$ are interpreted as functions between those sets.

We can now characterize when a concrete model has various type/kind structures by specifying their universal properties.

Definition 6.2. (1) For $B \in \mathcal{E}$, a thunk object $\mathsf{Thunk}\ B \in \mathcal{V}$ is given by a universal "force" morphism $J\,\mathsf{Thunk}\,B\,B$.
(2) For $A \in \mathcal{V}$, a return object $\mathsf{Ret}\ A \in \mathcal{E}$ is given by a universal "return" morphism $\mathsf{ret} \in J\,A\,\mathsf{Ret}\,A$.
(3) For I a set and $(B_i)_{i\in I}$ a family of objects of $\mathcal{E}$, a product $\&_{i\in I}B_i \in \mathcal{E}$ is given by a universal family of "projection" morphisms $\pi_i \in \mathcal{E}(\&_{i\in I}B_i, B_i)$ that are *distributive* in that for every $A \in \mathcal{V}$, the family of maps $\mathcal{J}\,A\,\pi_i$ is also universal.
(4) For I a set and $(A_i)_{i\in I}$ a family of objects of $\mathcal{V}$, a sum $\oplus_{i\in I}A_i \in \mathcal{V}$ is given by a universal family of "injection" morphisms $\sigma_i \in \mathcal{V}(A_i, \sum_{i\in I} A_i)$ that are *distributive* in that for every $B \in \mathcal{E}$, the family $\mathcal{J}\,\sigma_i\,B$ is also universal.
(5) A kind of value objects is a kind $\mathsf{VOb} \in \mathcal{K}$ given by a universal $\mathsf{VOb} \to \mathcal{V}_0$
(6) A kind of computation objects is a kind $\mathsf{COb} \in \mathcal{K}$ given by a universal $\mathsf{COb} \to \mathcal{E}_0$
(7) A function kind $k_o^{k_i} \in \mathcal{K}$ for $k_i, k_o \in \mathcal{K}$ is given by a universal $k_i \times k_o^{k_i} \to k_o$.
(8) A sigma kind $\sigma_{X:k}A(X)$ is given by a universal pair of projections $\pi_1 : \sigma_{X:k}A(X) \to k$ and $\pi_2 : (p : \sigma_{X:k}A(X)) \to A(\pi_1 p)$

We note that the distributivity condition for products (respectively sums) is automatically satisfied under the assumption that the model has all return (respectively thunk) objects.

One difference between the syntax and semantics is that the semantics has primitive notions of pure morphism (function between value objects) and linear morphism (morphism between computation objects), whereas the syntax has a more restrictive notion of values and we only give a primitive syntax for stacks in the CES semantics. The correct interpretation is that the morphisms between value objects $\mathcal{V}(A, A')$ correspond in the syntax to *thunkable* terms $x : A \vdash M : \mathsf{Ret}\ A'$ and similarly that computation morphisms $\mathcal{E}(B, B')$ correspond to *linear* terms $z : \mathsf{Thunk}\ B \vdash M : B'$, as described in [Munch-Maccagnoni 2014].

We can then define a Zydeco model to be a CBPV model that contains all the type structure in our syntax, except removing recursive data/codata and adding sigma kinds.

Definition 6.3. A Zydeco model is a concrete CBPV model with all thunk objects and return objects as well as

(1) Kinds of value objects, computation objects, function kinds and sigma kinds.
(2) Products indexed by 0 (modeling $\top$), 2 (modeling binary &), and objects of $\mathcal{K}$ (modeling impredicative $\forall$) as well as for any $B \in \mathcal{E}$ and $A \in \mathcal{V}$, a product $\&_{\_\in A} B$ (modeling the CBPV function type $\to$).
(3) Sums indexed by 0 (modeling the empty type), 2 (modeling +), and $\mathcal{V}_0$ and $\mathcal{E}_0$ (modeling impredicative $\exists$) as well as for any $A, A' \in \mathcal{V}$ a sum $\sum_{\_\in A} A'$ modeling the Cartesian product type $\times$.

6.2 Relative Monads, Algebras and the Fundamental Theorem

Next we introduce relative monads in a concrete CBPV model. Note that this definition of a relative monad requires no assumptions on type structure.

Definition 6.4. A *CBPV relative monad* consists of

(1) For each $A \in \mathcal{V}$, a type $MA \in \mathcal{E}$.
(2) For each $A \in \mathcal{V}$, an element $\eta_A : J\,A\,(MA)$
(3) For each $A, A' \in \mathcal{V}$ a function $-^\dagger : J\,A\,(MA') \to \mathcal{E}(MA, MA')$
(4) Satisfying $\eta^\dagger = \mathrm{id}$
(5) Satisfying $j^\dagger \circ \eta = j$
(6) Satisfying $(j^\dagger \circ k)^\dagger = j^\dagger \circ k^\dagger$

This has a fairly direct correspondence to our programming notion of monad in Zydeco: the η corresponds to the return, and the $-^\dagger$ operation to the bind operation with arguments reordered, but stated as families of set-theoretic functions rather than morphisms in CBPV, and with the linearity assumption of the bind operation instead encoded by requiring that the $-^\dagger$ operation return a morphism in $\mathcal{E}$. The Zydeco type is an *internalization* of the notion of a relative monad as a type, and one can verify that the elements of the interpretation of our Zydeco type of monad structures that satisfy the monad laws is isomorphic to a CBPV relative monad in this sense as long as $\mathcal{E}$ morphisms correspond precisely to linear morphisms. Note that because we are working internally to presheaves this notion of monads is automatically "strong" whether or not we internalize it, as we can define a version of $-^\dagger$ that works in an arbitrary context $-^\dagger : J\,A\,(\Gamma \to MA') \to \mathcal{E}(MA, \Gamma \to MA')$.

Definition 6.5. Let M be a CBPV relative monad for $\mathcal{M}$. An *algebra* for M consists of

(1) A carrier object $B \in \mathcal{E}$
(2) For every A a function $-^{\dagger_B} : J\,A\,B \to \mathcal{E}(MA, B)$
(3) Satisfying $j^{\dagger_B} \circ \eta = j$
(4) Satisfying $(j^{\dagger_B} \circ k)^{\dagger_B} = j^{\dagger_B} \circ k^\dagger$

Given two algebras $(B, -^{\dagger_B})$ and $(B', -^{\dagger'_B})$, an algebra *homomorphism* is a morphism $\phi : \mathcal{E}(B, B')$ that preserves the extension operation $\phi(j^{\dagger_B}) = \phi(j)^{\dagger'_B}$.

Definition 6.6. Given a relative monad M, an *Algebra kind* is a kind ALG $\in \mathcal{K}$ with a universal ALG $\to$ Alg(M).

Such an algebra kind can be implemented using iterated sigmas and extensional equality, but we will simply abstract over it directly.

This defines a category Alg(M), which we can then use to construct a new Zydeco model, the *algebra model*, for which the main functors are described in Figure 23.

Construction 6.1 (Fundamental Theorem of CBPV Relative Monads)**.** *Let M be a CBPV relative monad.*

We construct a new CBPV model, also called $Alg(M)$, as follows:

(1) *$\mathcal{V}, \mathcal{K}$ are given by the original $\mathcal{V}, \mathcal{K}$*
(2) *$\mathcal{E}$ is given by $Alg(M)$*
(3) *$\mathcal{C}$ is given by $\mathcal{C} \circ U$ where $U : Alg(M) \to \mathcal{E}$ is the forgetful functor.*

$Alg(M)$ always has all return objects, even if the original model does not, given by the free algebra $M\,A = (M\,A, -^{\dagger})$.

For the remaining connectives, $Alg(M)$ has...

(1) *A kind of value objects, function kinds and sigma kinds when the original model does.*
(2) *A kind of computation objects when the original type has a kind of algebras.*
(3) *Thunk objects when the original model does.*
(4) *The Zydeco sums whenever the original model does (trivially).*
(5) *The Zydeco product types whenever the original model does (see Lemma 6.2 below).*

In particular, if the original model is a Zydeco model with an algebra kind, then $Alg(M)$ is a Zydeco model.

Since all the Zydeco product types are given by a categorical product, this property follows from the following simple calculation for the forgetful functor, generalizing a well known property for ordinary monads[Arkor and McDermott 2023b]:

Lemma 6.2. *The forgetful functor $U : Alg(M) \to \mathcal{E}$ creates limits.*

Proof. Let $D : I \to \text{Alg}(M)$ be a diagram and let $L, (\pi_i)_{i \in I}$ be a limit cone for $U \circ D$. Then we construct an algebra structure on L by

$$j^{\dagger_L} = ((J\,A\,\pi_i\,j)^{\dagger_{D_i}})_{i \in I}$$

That is, the unique morphism such that $\pi_i \circ ((J\,A\,\pi_i\,j)^{\dagger_{D_i}})_{i \in I} = (J\,A\,\pi_i\,j)^{\dagger_{D_i}}$. The uniqueness condition of a limit is clearly satisfied because algebras form a subcategory. For the existence condition, let $\phi_i : \text{Alg}(M)((B, -^{\dagger_B}), D_i)$ be a family of algebra homomorphisms. Then we need to show that $(\phi_i)_{i \in I}$ is a homomorphism, which follows by a simple calculation

$$(\phi_i)_{i \in I} \circ j^{\dagger_B} = (\phi_i \circ j^{\dagger_B})_{i \in I} = (j^{\dagger_{D_i}})_{i \in I} = j^{\dagger_L}$$

□

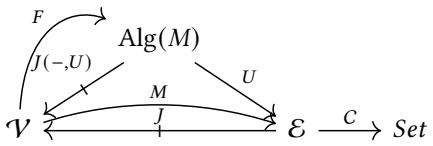

Fig. 23. Relating the Algebra Model to the original Model

6.3 Why CBPV Matters

We note that the CBPV separation into value and computation types, where value types have only "left adjoint" connectives besides Thunk and computation type connectives only have "right adjoint" connectives besides Ret , is essential for the fundamental theorem to hold. To demonstrate we consider two alternative systems where the corresponding fundamental theorem fails in that it is not a "self"-transformation or requires much more stringent requirements on the monad in question.

First, consider the situation in Moggi's original work: a strong monad on a Cartesian closed category. This is a "pure" λ calculus with a monad on it. We can take the category of algebras of this monad, but this will generally give back a Cartesian closed category. In general the category of algebras will model only the CBPV connectives, to get back a Cartesian category we would need the monad to strongly preserve Cartesian products[Kammar and Plotkin 2012]. If it only laxly preserves Cartesian products we get back a model of linear-non-linear logic[Benton and Wadler 1996].

Secondly, consider the enriched effect calculus, which adds several connectives to CBPV which break the "polarity" restriction, such as adding a coproduct of computation types $\oplus$ and a "tensor" $A \oslash B$ which is a dual of $A \to B$. In general given a CBPV relative monad, the category of algebras will not have coproducts even if the original $\mathcal{E}$ does, see Adamek and Koubek [1980] for conditions under which they exist.

7 FROM RELATIVE MONADS TO MONAD TRANSFORMERS

Programming explicitly with effects leads to issues of *interoperability*: can we call a subprocedure that uses monad M in a procedure using monad M'? As is well known, there is no arbitrary operation to take the "union" of effects by combining arbitrary monads together, such an operation is only supported by adding a restriction that the monad be given by an algebraic theory, which forms the basis for programming with "algebraic effects" [Plotkin and Power 2001].

One common way to modularly combine monads together without using algebraic theories is to use *monad transformers*. A monad transformer T is a sort of "higher-order" monad, it takes a monad M as input and outputs a monad $T(M)$ with a monad homomorphism $M \to T(M)$. We think of T as "adding" some effects to an arbitrary given other notion of effect provided by M. The homomorphism $M \to T(M)$ says that immediate subprocedures that use M can be used in a procedure that uses all of the effects of $T(M)$. If we think of relative monads operationally as providing some stack-based continuations, then monad transformers are a higher-order type of continuation constructors. If we take M to be the identity monad, then $T(\mathrm{Id})$ is an ordinary monad and we can think of the monad transformer T as a compositional version of the monad $T(\mathrm{Id})$.

Definition 7.1. A *monad homomorphism* h from $(M, \eta, -^\dagger)$ to $(M', \eta', -^{\dagger'})$ consists of a family of morphisms $h_A : \mathcal{E}(MA, M'A)$ that preserves the monad structure in that $h_A(\eta_A) = \eta'_A$ and $h_A(j^\dagger) = j^{\dagger'}$.

A *monad transformer* is a function $T : \mathrm{RMonad} \to \mathrm{RMonad}$ with a function taking any monad M and constructing a monad homomorphism $\mathrm{lift} : M \to T(M)$.

It is borne out by Haskell programming practice that seemingly all monads can be extended to a monad transformer, but the process of doing so is not automatic and involves some intuition based on experience. For instance, the exception monad $\mathrm{Exn}\, E\, A = E + A$ in a pure language can be generalized to the exception monad transformer $\mathrm{ExnT}\, M\, E\, A = M(E + A)$, but this takes some insight: the quite similar definition of $\mathrm{ExnT}\, M\, E\, A = E + M(A)$, for instance, is incorrect. Some general techniques for deriving monad transformers have been identified [Jaskelioff 2009] but none are fully general. The situation is improved in the setting of CBPV: *all* relative monads in CBPV can be extended to monad transformers. For instance, in the case of the relative exception monad $\mathrm{Exn}\, E\, A = F(E + A)$, we can derive the exception monad transformer simply by replacing all uses of F with the monad parameter $\mathrm{ExnT}\, M\, E\, A = M(E + A)$. Intuitively this is because all CBPV constructions work with an arbitrary notion of ambient effect, which can be instantiated even with user-provided effects. The proof formalizes this intuition using the algebra translation from Section 6 to reinterpret a monad in the model of algebras of the "generic" monad. Externalizing the construction constructs a monad transformer.

This property is not just convenient for CBPV programming, but can be used to derive monads in "pure" like Haskell based on λ calculus since pure languages can be seen as a degenerate model of CBPV with no effects. Using this inclusion, we get that all relative monad transformers in CBPV induce ordinary monad transformers in the language of pure λ calculus. This provides a systematic, if not automatic, method for deriving a monad transformer in a pure language from a monad: first find a relative monad in CBPV that coincides with the original monad in the pure model, and then automatically derive the definition of the monad transformer. This at least provides some guidance to the programmer, for instance in the exception monad case above, the incorrect definition of $E + M(A)$ doesn't pass kind-checking in CBPV!

If the CBPV model has return objects, then Ret is a relative monad and further, it is the *initial* relative monad, for any monad M there is a unique relative monad homomorphism from Ret to M using the algebra structure all computation types carry. This leads to the following observation, which generalizes a similar property for ordinary monads:

Construction 7.1. *Given M a relative monad in $Alg(M_b)$, we can construct a relative monad UM in the original model with a monad homomorphism $M_b \to UM$.*

Proof. As suggested, UM is constructed by taking the monad structure of M and simply forgetting the algebra/homomorphism properties. The homomorphism $M_b \to UM$ is given by similarly forgetting the additional algebra homomorphism conditions in the monad homomorphism $\mathrm{Ret} \to M$. □

With this, we can give our monad transformer construction:

Construction 7.2. *Any monad M definable in an arbitrary Zydeco model extends to a relative monad transformer $T(\mathrm{Ret}) \cong M$.*

Proof. For an input monad M_b, we construct the algebra model $\mathrm{Alg}(M_b)$. We then interpret M in $\mathrm{Alg}(M_b)$ and apply Construction 7.1 to get a monad we call $T(M_b)$ which comes with a monad homomorphism $M_b \to T(M_b)$. If the input monad M_b is Ret , then the algebra model is equivalent to the original, and so interpreting M in Alg(Ret) results in an equivalent monad $T(\mathrm{Ret}) \cong M$. □

Finally, we can apply this to a "pure" F_ω style language based on the following:

Definition 7.2. An *depolarized CBPV model* is a polymorphic concrete CBPV model where $\mathcal{E} = \mathcal{V}$ and $C : \mathcal{V} \to Set$ is the inclusion. A *concrete F_ω model* is a Zydeco model that is a pure CBPV model.

Observe that in a depolarized model J is just the Hom profunctor of $\mathcal{V}$. Additionally, a relative monad in a depolarized model is just an ordinary (strong) monad, and similarly a relative monad transformer is just an monad transformer. This makes the following simply an instance of Construction 7.2

Corollary 7.1. *Any monad M definable in an arbitrary Zydeco model defines an ordinary monad transformer in an F_ω model.*

8 DISCUSSION AND FUTURE WORK

8.1 Implementing the Algebra Translation

Currently there is a slight mismatch between the fundamental theorem we proved in Section 6 and the Zydeco language: Zydeco does not currently support Σ kinds and so it is not possible to directly encode the algebra translation homomorphically in the way we described at least when quantification over computation types is used. In Zydeco we can define e.g., a relative continuation monad parameterized by R as $\mathrm{Cont}\, A\, R = U(A \to R) \to R$. The algebra translation would translate this definition to something equivalent to $\mathrm{Cont}\ (A : VTy)\ (R : SigmaR0.(RAlg : AlgebraR0M) = U(A \to R.1) \to R.1$, where M is the monad parameterizing the translation, but here we see that while the original definition is valid using only quantification over types, the definition produced by the algebra translation quantifies over the *implementation* of an algebra, i.e., a type dependent on a value. In practice, we carry out this process by hand, for example when deriving the monad transformer from a given monad. To circumvent the need for sigma kinds we can typically move the algebra parameter from the type definition to all of the uses, i.e., in the above definition we only use $R.1$ in the definition of the carrier. This technique can likely be adapted to a systematic translation that stays in the system F_ω fragment, but using Σ kinds would lead to a simpler translation at higher kind since using the Σ-kind based translation, all kind constructors besides CType can be homomorphically translated.

An alternative to a meta-theoretic algebra translation would be to use Haskell-style typeclasses for relative monad and algebra, where `do` notation would be overloaded to use the algebra structure of the type of the expression. To get the full convenience of this translation would require a mix of features currently supported by Haskell (unification-based type inference, typeclasses, deriving) but which would require considerable engineering effort to be added to Zydeco. It is also not clear how this encoding would work at higher kinds.

Another avenue for future work would be to extend Zydeco to support dependent CBPV, and this also causes issues for an automated algebra translation. The reason is that in order to support decidable type checking we would support an intensional type theory, but constructive mathematical proofs like the one we gave are only directly translatable to *extensional* type theory. Tackling such issues would require adapting techniques used in e.g., the weaning translation for ∂-CBPV [Pédrot and Tabareau 2017, 2019]. In particular, the weaning translation is an adaptation of the algebra semantics of CBPV for an ordinary monad to the setting of intensional type theory, so our relative monad algebra semantics could possibly be combined to generalize this to a "relative weaning translation".

8.2 Applications to Verified Compilation

In this work, for concreteness we have focused on a stack-based calculus, but much of our interest in this topic is on potential applications to verified compilation, where low-level representation

decisions such as the representation of continuations and stack frames must be formalized and code verified to use this representation. We argue that there is not quite as large a gap between the CBPV-based calculus we have used here and low-level compiler IRs. Firstly, the CBPV calculus is closely related to common functional compiler IRs such as ANF and CPS [Appel 1992; Flanagan et al. 1993], and additionally to SSA form, the most commonly used compiler IR in industry today. The most thorough result about this relationship proves an isomorphism between focused CBPV terms and SSA control-flow graphs [Garbuzov et al. 2018]. However these intermediate representations typically abstract from explicit stack manipulation, and so CBPV is somewhat more general in its ability to describe stack structure in its type system. In this regard it is more similar to explicit stack-manipulating IRs such as the Stack-based Typed Assembly Language [Morrisett et al. 2003]. Stack-based TAL includes, like CBPV, a second "kind" of type, called stack types, and indeed their work includes examples of implementing exceptions using stack types that inspired some of our examples in Zydeco. For instance, they give encodings of double-barreled continuations that, other than specifying additional details such as the representation of closures and precisely which registers are used correspond directly to our encoding in CBPV. A major difference in our work is our use of *coinductive* computation types to encode stack-walking, which would correspond dually to a kind of *inductive* stack type in stack-based TAL.

8.3 Relative Comonads

An obvious direction for future work would be to consider relative *comonads* in CBPV, which would dually be type constructors $CType \to VType$ with extra structure, with U being the prototypical example, just as F is the prototypical relative monad. In the same way that relative monads abstract the structure of continuations implemented on the stack to allow for additional "effects", relative comonads would abstract the structure of *closures* to allow for additional "coeffects" such as accessing metadata for the closure or availability of a destructor for an object. It is however not clear what the analog of the fundamental theorem of relative monads for comonads would be as categories of coalgebras would not necessarily have Cartesian products: this might require generalizing the value of types of CBPV to be linear types or adding further restrictions to the notion of comonad considered.

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2429069.2429075

Jiri Adamek and Vaclav Koubek. 1980. Are colimits of algebras simple to construct? *Journal of Algebra* 66, 1 (1980), 226–250. https://doi.org/10.1016/0021-8693(80)90122-2

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.* 342, 1 (2005), 149–172. https://doi.org/10.1016/J.TCS.2005.06.008

Daniel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures*. 297–311.

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

Nathanael Arkor and Dylan McDermott. 2023a. The formal theory of relative monads. arXiv:2302.14014 [math.CT]

Nathanael Arkor and Dylan McDermott. 2023b. Relative monadicity. arXiv:2305.10405 [math.CT]

Steve Awodey. 2018. Natural models of homotopy type theory. *Math. Struct. Comput. Sci.* 28, 2 (2018), 241–286. https://doi.org/10.1017/S0960129516000268

P. N. Benton and Philip Wadler. 1996. Linear Logic, Monads and the Lambda Calculus. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society,

420–431. https://doi.org/10.1109/LICS.1996.561458

David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2022. Data-Codata Symmetry and its Interaction with Evaluation Order. https://doi.org/10.48550/arXiv.2211.13004 arXiv:2211.13004 [cs].

Pierre-Louis Curien, Marcelo P. Fiore, and Guillaume Munch-Maccagnoni. 2016. A theory of effects and resources: adjunction models and polarised calculi. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 44–56. https://doi.org/10.1145/2837614.2837652

Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 119)*, Dan R. Ghica and Achim Jung (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:23. https://doi.org/10.4230/LIPIcs.CSL.2018.21

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *Comput. Surveys* 54, 5 (June 2022), 1–38. https://doi.org/10.1145/3450952 arXiv:1908.05839 [cs].

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2014. The enriched effect calculus: syntax and semantics. *J. Log. Comput.* 24, 3 (2014), 615–654. https://doi.org/10.1093/LOGCOM/EXS025

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. 2018. Structural Operational Semantics for Control Flow Graph Machines. *CoRR* abs/1805.05400 (2018). arXiv:1805.05400 http://arxiv.org/abs/1805.05400

Mauro Jaskelioff. 2009. Modular Monad Transformers. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 64–79. https://doi.org/10.1007/978-3-642-00590-9_6

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 349–360. https://doi.org/10.1145/2103656.2103698

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Paul Blain Levy. 2001. *Call-by-Push-Value.* Ph. D. Dissertation. Queen Mary, University of London, London, UK.

Paul Blain Levy. 2017. Contextual isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 400–414. https://doi.org/10.1145/3009837.3009898

Paul Blain Levy. 2019. What is a Monoid? (2019). https://conferences.inf.ed.ac.uk/ct2019/slides/83.pdf Category Theory 2019.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514. https://doi.org/10.1145/3009837.3009897

David B. MacQueen. 1984. Modules for Standard ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr. (Eds.). ACM, 198–207. https://doi.org/10.1145/800055.802036

Maria Emilia Maietti. 2005. Modular correspondence between dependent type theories and categories including pretopoi and topoi. *Math. Struct. Comput. Sci.* 15, 6 (2005), 1089–1149. https://doi.org/10.1017/S0960129505004962

John C. Mitchell and Robert Harper. 1988. The Essence of ML. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 28–46. https://doi.org/10.1145/73560.73563

Rasmus Ejlers Møgelberg and Alex Simpson. 2009. Relational Parametricity for Computational Effects. *Log. Methods Comput. Sci.* 5, 3 (2009). http://arxiv.org/abs/0906.5488

Eugenio Moggi. 1991. Notions of computation and monads. *Inform. And Computation* 93, 1 (1991).

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. 2003. Stack-based typed assembly language. *J. Funct. Program.* 13, 5 (2003), 957–959. https://doi.org/10.1017/S0956796802004446

Guillaume Munch-Maccagnoni. 2014. Models of a Non-associative Composition. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). 396–410.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498670

Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation using Linking Types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2023, Seattle, WA, USA, 4 September 2023*, Youyou Cong and Pierre-Évariste Dagand (Eds.). ACM, 14–28. https://doi.org/10.1145/3609027.3609405

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *IEEE Symposium on Logic in Computer Science (LICS), Reykjavik, Iceland.*

Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.* 4, POPL, Article 58 (dec 2019), 28 pages. https://doi.org/10.1145/3371126

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, 71–84. https://doi.org/10.1145/158511.158524

Gordon D. Plotkin and John Power. 2001. Semantics for Algebraic Operations. In *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 45)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference on - ACM '72*, Vol. 2. ACM Press, Boston, Massachusetts, United States, 717–740. https://doi.org/10.1145/800194.805852

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

Hayo Thielecke. 2001. Comparing Control Constructs by Double-barrelled CPS Transforms. *Electronic Notes in Theoretical Computer Science* 45 (2001), 433–447. https://doi.org/10.1016/S1571-0661(04)80974-5 MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, 61–78. https://doi.org/10.1145/91556.91592

Mitchell Wand. 1980. Continuation-Based Program Transformation Strategies. *J. ACM* 27, 1 (1980), 164–180. https://doi.org/10.1145/322169.322183

$\boxed{\Sigma \vdash G \dashv \Delta}$ G extracts Δ under type declaration context Σ

$$\frac{\text{data } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \text{end} \in \Sigma \qquad \Delta = \{X \mapsto (K_k)^* \to \mathsf{VType}\}}{\Sigma \vdash \text{data } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \text{end} \dashv \Delta}\ [\textsc{DataExtract}]$$

$$\frac{\begin{array}{c}\text{codata } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ : B_i\ )^*\ \text{end} \in \Sigma \\ \Delta = \{X \mapsto (K_k)^* \to \mathsf{CType}\}\end{array}}{\Sigma \vdash \text{codata } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ : B_i\ )^*\ \text{end} \dashv \Delta}\ [\textsc{CoDataExtraction}]$$

Fig. 24. Rules on Declaration Extraction

$\boxed{\Sigma \vdash G\ \text{valid}}$ G is valid under declaration context Σ

$$\frac{\Delta_\Sigma,\ (X_k : K_k)^* \vdash T_{ij} : \mathsf{VType}}{\Sigma \vdash \text{data } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ C_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ )^*\ \text{end valid}}\ [\textsc{DataValid}]$$

$$\frac{\Delta_\Sigma,\ (X_k : K_k)^* \vdash T_{ij} : \mathsf{VType} \qquad \Delta_\Sigma,\ (X_k : K_k)^* \vdash T_i : \mathsf{CType}}{\Sigma \vdash \text{codata } X\ (X_k : K_k)^*\ \text{where}\ (\ |\ .D_i\ (X_{ij} : K_{ij})^*\ (A_{ij'})^*\ : B_i\ )^*\ \text{end valid}}\ [\textsc{CoDataValid}]$$

Fig. 25. Rules on Declaration Validation

A TYPE DECLARATION RULES OF ZYDECO

The declaration extraction pass extracts the kinding information of all the new types defined in the declaration as shown in Figure 24.

B MONAD LAWS

We verify verify the monad laws on the exception monads satisfying the canonicity condition. The right unital law is in fact satisfied without the extra conditions;

$$\begin{aligned} & !\ \text{bind}\ \{\ !\ \text{return}\ a\ \}\ f \\ \equiv\ & !\ \text{return}\ a\ .\text{kont}\ f \\ \equiv\ & !\ f\ a \end{aligned}$$

And the left unital law can be verified if we apply the canonicity condition on $!\, m$.

$$
\begin{aligned}
& !\ \mathsf{bind}\ m\ \mathsf{return} \\
\equiv\ & !\ m\ .\mathsf{kont}\ \mathsf{return} \\
\equiv\ & \mathsf{do}\ x \leftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \rightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \rightarrow !\ \mathsf{return}\ a \\
& \mathit{end} \\
& .\mathsf{kont}\ \mathsf{return} \\
\equiv\ & \mathsf{do}\ x \leftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \rightarrow !\ \mathsf{fail}\ e\ .\mathsf{kont}\ \mathsf{return} \\
& \mid + \mathit{Right}(a) \rightarrow !\ \mathsf{return}\ a\ .\mathsf{kont}\ \mathsf{return} \\
& \mathit{end} \\
\equiv\ & \mathsf{do}\ x \leftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \rightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \rightarrow !\ \mathsf{return}\ a \\
& \mathit{end} \\
\equiv\ & !\ m
\end{aligned}
$$

As for the associativity law, we observe

$$
\begin{aligned}
!\ \mathsf{bind}\ \{\ !\ \mathsf{bind}\ m\ f\ \}\ g\ &\equiv !\ m\ .\mathsf{kont}\ f\ .\mathsf{kont}\ g \\
!\ \mathsf{bind}\ m\ \{\ \mathsf{fn}\ x \rightarrow !\ \mathsf{bind}\ \{\ !\ f\ x\ \}\ g\ \}\ &\equiv !\ m\ .\mathsf{kont}\ \{\ \mathsf{fn}\ x \rightarrow !\ f\ x\ .\mathsf{kont}\ g\ \}
\end{aligned}
$$

By applying the canonicity condition on $!\, m$ and using the same reasoning as in the left unital law, we have for the left side

$$
\begin{aligned}
& !\ m\ .\mathsf{kont}\ f\ .\mathsf{kont}\ g \\
\equiv\ & \mathsf{do}\ x \leftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \rightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \rightarrow !\ \mathsf{return}\ a\ .\mathsf{kont}\ f\ .\mathsf{kont}\ g \\
& \mathit{end} \\
\equiv\ & \mathsf{do}\ x \leftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \rightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \rightarrow !\ f\ a\ .\mathsf{kont}\ g \\
& \mathit{end}
\end{aligned}
$$

and the right side

$$
\begin{aligned}
& !\ m\ .\mathsf{kont}\ \{\ \mathsf{fn}\ x\ \longrightarrow !\ f\ x\ .\mathsf{kont}\ g\ \} \\
\equiv\ & \mathsf{do}\ x \longleftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \longrightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \longrightarrow !\ \mathsf{return}\ a\ .\mathsf{kont}\ \{\ \mathsf{fn}\ x\ \longrightarrow !\ f\ x\ .\mathsf{kont}\ g\ \} \\
& \mathit{end} \\
\equiv\ & \mathsf{do}\ x \longleftarrow !\ m\ .\mathsf{done}\ ; \\
& \mathsf{match}\ x \\
& \mid + \mathit{Left}(e) \longrightarrow !\ \mathsf{fail}\ e \\
& \mid + \mathit{Right}(a) \longrightarrow !\ f\ a\ .\mathsf{kont}\ g \\
& \mathit{end}
\end{aligned}
$$

Finally, the linear bind law trivially holds because M is syntactically restricted to be used for only once in the computation. The only surrounding computation of M is a do-binding, which preserves linearity. Therefore, we conclude that all four monad laws are satisfied with the canonicity property.