Lecture 18: Extensions to STT, Monoid Actions

Lecturer: Max S. New
Scribe: Michael Xi

March 20, 2023

Abstract

We attempt to add more effects to our programming language.

1 An Aside

1.1 The Disjunction Property

We recall the disjunction property in IPL. That is,

$$\text{if } \cdot \vdash A \vee B \text{ is provable, then either } \cdot \vdash A \text{ is provable, or } \cdot \vdash B \text{ is provable}$$

We then wish to prove a similar property holds for our representation of logical disjunctions in our category $\mathcal{G}$. This can also be proven by our gluing proof, by a similar reason to the ones used to prove our boolean semantics. In particular we have the stronger result that if a term of sum type $C + D$ can be proved from an empty context, then $M = i_1 M'$ or $M = i_2 M'$ (but not both). Here M' corresponds to your proof of A or B as terms in IPL.

1.2 The Story So Far

Insofar we've covered a simple form of logic in IPL, and an example of a pure functional programming language in STT. Of course, polymorphism, data abstraction, etc. are found in pure functional programming languages but out of scope for this course.

2 "Real(?)" Programming Languages

We now consider what makes a programming language "real". Some features suggested are:

1. Memory, and the ability to modify it
2. "Side Effects", such as being able to print and write to files

3. Turing completeness, the ability to simulate any Turing machine, or equivalently to compute every Turing-computable function
4. "Going wrong", the ability to crash
5. Concurrent execution

Concurrency is unfortunately "too real" for this semester of class, but we will discuss the others.

3 Extending our Language

Now that we have some candidates, we'd like to extend our existing language.

3.1 Semi-Colon, Sequentiality

One might say the semi-colon ;, is the critical difference between a "real" programming language and math. We consider the following:

$$M; N$$

Which is equivalent to saying "do M, then, do N". We now wish to add the concept of a semi-colon, that is, sequentiality, to our pure functional programming language.

3.2 Printing, Logging

We would like to consider a program that might be able to "print" in some fashion, perhaps by writing 0's and 1's (as binary). This we accomplish by adding two new forms to our language, called **beep** and **boop**. They act by sending a "beep" or "boop" (0 or 1) to our printer, where timing is ignored, and we assume all 0's and 1's sent to the printer are printed at once. That is:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{beep}; M : A}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{boop}; M : A}$$

We recall by our set-theoretic semantics, and our proof by gluing, that something of boolean type $(1 + 1)$, a program, was either equal to $i_1()$ (true), or $i_2()$ (false). However should this be true, our "beeps" and "boops" would be useless, as they would always equal either $i_1()$ or $i_2()$, and thus under our interpretation signal the printer not to print anything.

4 An Attempt at Canonicity

We'd like to have a similar canonicity theorem to STT, where, using the equational theory, we are able to classify all the terms of type $1 + 1$.

A natural formulation would be that all such terms are equal to a finite string of beeps and boops followed by a "bit" equal to either 1, or 2, the elements of type $1 + 1$.

Our **specification** is then: $\forall(\cdot \vdash M : 1 + 1)\ \exists! s \in \{\texttt{beep}, \texttt{boop}\}^*\ \exists!\text{bit} \in \{1, 2\}$ such that $M = S; \text{bit}$. That is, all terms are equivalent to a finite string of beeps and boops followed by a bit, either valued 1 or 2 ($i_0()$, $i_1()$), but that all such distinct strings of beeps and boops followed by a bit are not equal.

That is, $M = i_1()$, or $M = i_2()$, or $M = \texttt{beep}; i_1()$, or, ..., but cannot be equal to two of these at once.

4.1 Sum Types

Unfortunately that under some natural assumptions, the currently described system will not be able to meet this specification.

We consider a case statement. It is natural to require that when performing a case analysis on something that beeps or boops, the term overall should beep or boop and then case on the remaining term. That is, we want the **Strictness of Case$_+$**:

$$\begin{aligned} & \text{case beep}; i_1() \begin{cases} i_1 x_1 \implies M_1 \\ i_2 x_2 \implies M_2 \end{cases} \\ = \quad & \text{beep; case} i_1() \begin{cases} i_1 x_1 \implies M_1 \\ i_2 x_2 \implies M_2 \end{cases} \\ =_{\beta+} \quad & \texttt{beep}; M_1 \end{aligned}$$

But this is already enough to make beeping trivial:

Lemma 1. *If case is strict, then* $\texttt{beep}; i_1() = i_1()$

Proof. Note

$$x : 1 + 1 \vdash \text{case } x \begin{cases} i_1 x_1 \implies i_1() \\ i_2 x_2 \implies i_1() \end{cases} =_{\eta+} i_1() \tag{1}$$

By the principle of subsutitivity, if an equation is proved true with a free variable of give type in it, that variable may be instantiated with another of the same type and the equation will still hold (primitive or admissable depending on formulation).

So we consider beep; $i_1() : 1 + 1$ and substitute, yielding

$$\text{case}_+(\texttt{beep}; i_1) \begin{cases} i_1 x_1 \implies i_1() \\ i_2 x_2 \implies i_1() \end{cases} = i_1() \tag{2}$$

Then, by strictness,

$$\texttt{beep}; (\text{case}_+(i_1) \begin{cases} i_1 x_1 \implies i_1() \\ i_2 x_2 \implies i_1() \end{cases}) = i_1() \tag{3}$$

However, by β-reduction,

$$(\text{case}_+(i_1) \begin{cases} i_1 x_1 \implies i_1() \\ i_2 x_2 \implies i_1() \end{cases}) = i_1() \tag{4}$$

So we conclude that beep; $i_1 = i_1$. □

So if the case statement is strict, given all the connectives (in particular the sum and unit), then our specification above fails as a term cannot be both equivalent to beep; $i_1()$ and $i_1()$ at the same time.

4.2 Function Types

We note we rely on η for sums, the principle of subsutitivty for equality, and the strictness of case_+ to arrive at our result above. So while we might suggest there is something wrong with the sum type, it happens that a similar issue arises when we consider the function type.

Let us consider a general semicolon, where for terms $M; N$, we do M, then N, and can also use the result of M in N. That is, for immutable variable x,

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \texttt{var}\ x = M; N : B}$$

However then, for $(\texttt{var}\ x = \texttt{beep}; N); M$, we expect this to beep first, then do N, and then do M, providing a similar commutivity to our previous examples with case. That is,

$$(\texttt{var}\ x = (\texttt{beep}; N); M) = (\texttt{beep}; (\texttt{var}\ x = M; N))$$

We call this the **Strictness of Var**.

Additionally if M is just another variable, it is natural to say that var is equivalent to a substitution:

$$(\texttt{var}\ x = y; N) = N[y/x]$$

Consider now, $f : A \to B \vdash \texttt{var}\ g = (\texttt{beep}; f); i_1() : 1 + 1$. Certainly this should be beep first, so by above, we have the following

$$\begin{aligned} f : A \to B \vdash & \texttt{var}\ g = (\texttt{beep}; f); i_1() : 1 + 1 \\ & \texttt{beep}; (\texttt{var}\ g = f; i_1()) \\ & \texttt{beep}; i_1()[f/g] \\ & \texttt{beep}; i_1() \end{aligned}$$

Note beep;$f : A \to B$. By η-expansion, then,

$$f : A \to B \vdash \texttt{var}\ g = (\texttt{beep}; f); i_1() : 1 + 1$$
$$\texttt{var}\ g = (\lambda x.(\texttt{beep}; f)x); i_1()$$

However we'd expect this unused lambda to not do anything, save being assigned. That is,

$$(\texttt{var}\ y = (\lambda x.M); N) = (N[\lambda x.M/y])$$

So as g is not used in $i_1()$

$$(\texttt{var}\ g = (\texttt{beep}; f); i_1()) = (\texttt{var}\ g = (\lambda x.(\texttt{beep}; f)x); i_1()) = i_1()$$

Because of this we run into a similar problem as with sum types. In particular $\texttt{var}\ g = (\texttt{beep}; f); i_1()$ is equivalent to $\texttt{beep}; i_1()$ by applying the strictness of var. But it is also equivalent to $i_1()$ by η-rules followed by the above rule for lambdas. So beep; $i_1() = i_1()$.

The culprits this time around are η-rules for functions, strictness of var, and substitution properties involving assignment to an immutable operator.

4.3 Crashing

We now wish to crash. To do this we define a term exit in absence of semicolon, which crashes our program,

$$\frac{}{\cdot \vdash \texttt{exit} : 0}$$

We do lose a connection to logic, as we may prove 0 from an empty context, and thus conclude anything. Consistency of the IPL logic is not strictly maintained, but we would like to still maintain consistency of the equational theory.

We recall we wished to prove that,

$$\frac{}{\cdot \vdash i_1() = i_2() : 1 + 1}$$

did not hold. If it did, it would break canonicity, and mean our programs are no longer useful as a specification for producing bits (as the bits would be indistinguishable).

However, by the η-rules for 0,

$$\frac{\cdot \vdash \texttt{exit} : 0}{\cdot \vdash i_1() = i_2() : 1 + 1} 0\eta'$$

Representing an implicitly weakening of the second line, noting the original 0η rule indicates,

$$\frac{}{\texttt{exit} : 0 \vdash i_1() = i_2() : 1 + 1} 0\eta$$

So exit, together with the 0η-rule, the entire language becomes trivial as we cannot distinguish between $i_1()$ and $i_2()$.

The culprit this time is just the η-rule for 0.

4.4 Our Culprits

So we note η-rules have been culprits for each break of consistency. So, to describe a programming language by an equation theory, as with STT, but including effects, we cannot continue to use an extension to STT. We must lose some of:

- η rules
- general substitutivity rules
- strictness rules

In eager/call-by-value languages, we lose η for functions and products, and also general substitutivity. In lazy/call-by-name languages, we lose η for sums and empty, and also some strictness properties. A third way, which most closely matchers compiler intermediate languages, keeps η, substitutivity and strictness, but at the cost of changing the underlying judgments from STT to incorporate new kinds of terms.

We will now attempt to "discover" this new type theory by working backwards from our specification and its semantics.

5 Working Backwards

Let us conider our canonicity principle for our extension with beeps and boops.

For exit, similarly, we have a canonicity result, indicating, $\forall(\cdot \vdash M : 1 + 1)$, then $M = i_1()$, or $M = i_2()$ or $M = exit$, but not more than one. That is, we may exit or produce a value but not both.

Similar canonicity statements exist for other formulations, such as mutable state. For one mutable cell, with read and write terms,

$$\frac{}{\cdot \vdash \texttt{read} : 1 + 1}$$

$$\frac{M : 1 + 1}{\cdot \vdash \texttt{write}\ M : 1}$$

Then, $\forall(\cdot \vdash M : 1 + 1)$,

$$M = \mathrm{case}_+ \texttt{read} \begin{cases} i_1 x_1 \implies \texttt{write}(b_1); b_1' \\ i_2 x_2 \implies \texttt{write}(b_2) b;_2 \end{cases}$$

That is, there exist four unique booleans, where the program reads the initial state and writes a final state based on the initial state and returning a boolean result.

Given these canonicity statements we wish to work backwards to determine a semantic way to describe these computational phenomena, and thus derive our language.

5.1 Monoid Actions

We return to our language of beeping and booping. We recall for any term of type A, we have an operation where we can add a beep or boop. The associated canonicity result is that every program is a string of beeps and boops, followed by a boolean. That is, we wish to take a string of these beeps and boops, output those strings, then produce the program.

In the beep boop language, the set of terms of type $1 + 1$ has a monoid action of $\{\text{beep, boop}\}^*$. That is,

$$\frac{s \in \{\text{Beep, Boop}\}*, \Gamma \vdash M : 1 + 1}{\Gamma \vdash S \cdot M : 1 + 1}$$

This action is defined by recursion, where ++ denotes string concatenation:

$$S \cdot M = \begin{cases} M & \text{if S} = \text{""} \\ S' \cdot \texttt{beep}; M & \text{if S} = \text{S'++beep} \\ S' \cdot \texttt{boop}; M & \text{if S} = \text{S'++boop} \end{cases}$$

This is a monoid action as for strings S, S' and term M, $(S{+}{+}S') \cdot M = S \cdot (S' \cdot M)$ by casing on S', and the identity "" is the identity by definition.

We recall our set-theoretic semantics underlying our gluing argument for our canonicity result for STT. What we can do to prove our more general canonicity results are to make more general semantics, replacing sets and functions.

In our case we have the set of $\{\text{beep, boop}\}^*$-actions, and the set of closed terms carrying such an action ($\cdot \vdash M : 1 + 1$). Ultimately we wish to conclude that there is a purely semantic set of closed terms with a monoid action that is isomorphic to the set of monoid actions.

Definition 1. *For any set X, and monoid M, define the* ***Free M-action*** *from X as* $F_{M\text{-}Action}X$.

1. $(F_{M\text{-}Action}X) = |M| \times X$, *the underlying set of M, times X*
2. $\_ \otimes \_ : |M| \times |F_{M\text{-}Action}X| \to F_{M\text{-}Action}X$, *defined by* $m \otimes (n, x) := (m \cdot n, x)$

For $M = \{\text{beep, boop}\}^*$, $X = \{1, 2\}$, then $F_{M-\text{Action}}X = \{\text{beep, boop}\} * \times \{1, 2\}$. Our canonicity result would follow if we could show $F_{M-\text{Action}}X \cong \{\cdot \vdash M : 1 + 1\}$, that the monoid action is isomorphic to the monoid action of closed terms of type $1 + 1$.

We wish to establish a bijection of sets, and isomorphism of actions.

5.2 Is This a Functor?

We note $F_{M-\text{Actions}} : \text{Sets} \to \text{MAct}$ where MAct is the category of M-actions. However we would like to know if this is a functor, and moreover if it has the universal property. Consider the following:

Lemma 2. *$F_{M-Actions} : Sets \to MAct$ is a functor.*

Proof. Given a function $f : X \to Y$, we wish to define a function $F_{M-\text{Actions}}f : F_{M-\text{Actions}}X \to F_{M-\text{Actions}}Y$.

Let $F_{M-\text{Actions}}f(m, x) = (m, fx)$. We note this preserves identity as then $fx = x$, and composition as

$$\begin{aligned}(F_{M-\text{Actions}}f \circ g)(m, x) &= \\ (m, f(g(x))) &= \\ (F_{M-\text{Actions}}f)(m, g(x)) &= \\ (F_{M-\text{Actions}}f) \circ (F_{M-\text{Actions}}g)(m, x)\end{aligned}$$

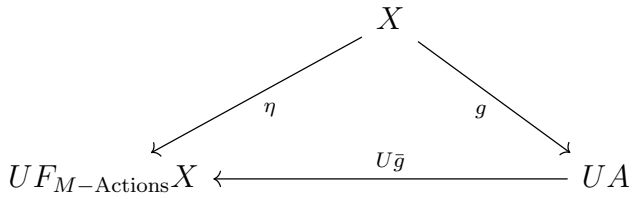
We do have to check this is in fact an equivariant function. Consider

$$F_{M-\text{Actions}}f(m \cdot (n, x)) = (m \cdot n, fx) = m \cdot (n, fx) = m \cdot F_{M-\text{Actions}}f(n, x)$$

So it is in fact equivariant. So $F_{M-\text{Actions}}$ is a functor. □

We claim this M-action has a universal property, which is why we call it a "free" M-action.

1. $U : \text{MAct} \to \text{Set}$, such that $U(X, \otimes) = X$ and $Uf = f$ (forgetful functor to underlying set of monoid action)
2. $\exists \eta : X \to UF_{M-\text{Actions}}X$, such that $\eta(x) = (e, x)$ where e is the neutral element of the monoid
3. $\forall G : X \to UA$, $\exists ! \bar{g} : F_{M-\text{Actions}}X \to A$, a homomorphism (equivariant function), such that the following diagram commutes:



So, for each X, we have a universal property for $F_{M-\text{Actions}}X$, where $F_{M-\text{Actions}}X$ is the free monoid action. This is in fact a common property of universal property, where we can specify a functor F by specifying that for each object it has a universal property we can define similarly in terms of U. F, U satisfying these properties are called **adjoint functors**.