

Lecture 22: Call-by-push-value

Lecturer: Max S. New

Scribe: Eric Zhao

April 03, 2023

1 Review of beep-boop

Recall the language of beeps and boops we defined previously, where we had the following two constructors:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{beep}; M : A} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{boop}; M : A}$$

Our expected canonicity result for this language stated that for all closed terms $\cdot \vdash M : \text{Bool}$ (where Bool is $1 + 1$), $M = (\text{beep}; | \text{boop};)^*(\iota_1() | \iota_2())$.

Remember that we can view this language in terms of monoid actions, specifically with the monoid \mathcal{M} of strings over $\{\text{beep}; , \text{boop}; \}$. Recall that for any monoid, there is an adjunction:

$$\begin{array}{ccc} & F_{\mathcal{M}\text{-Act}} & \\ \text{Set} & \begin{array}{c} \curvearrowright \\ \perp \\ \curvearrowleft \end{array} & \mathcal{M}\text{-Act} \\ & U & \end{array}$$

where U takes the monoid to its underlying set and $F_{\mathcal{M}\text{-Act}}$ is the free monoid action:

$$F_{\mathcal{M}\text{-Act}}X := (|\mathcal{M}| \times X, m \cdot (n, x) = (m \cdot n, x))$$

This is the minimal way to equip a set with a monoid action, and the unit of the adjoint is $\eta : X \rightarrow UF_{\mathcal{M}\text{-Act}}X$ such that $\eta x := (e, m)$.

This gives us the ability to interpret terms in a semantics of monoid actions:

$$\cdot \vdash M : \text{Bool} \rightsquigarrow \llbracket M \rrbracket \in UF_{\mathcal{M}\text{-Act}}(\{\text{true}, \text{false}\}) = M \times \{\text{true}, \text{false}\}$$

2 Eager vs. lazy semantics

Let's look at the interpretation of terms of other types. Consider

$$\cdot \vdash M : \text{Bool} \times \text{Bool}$$

We might expect the following evaluation result

$$UF_{\mathcal{M}\text{-Act}}(\{\text{true}, \text{false}\} \times \{\text{true}, \text{false}\}) = \mathcal{M} \times (\{\text{true}, \text{false}\} \times \{\text{true}, \text{false}\})$$

But this is strange from the perspective of the categorical semantics, since we have in the past always interpreted the product in $\text{Bool} \times \text{Bool}$ as a product in the category where we are taking the semantics.

For $A_1, A_2 \in \mathcal{M}\text{-Act}$, $A_1 \times A_2 = (|A_1| \times |A_2|, m \cdot (a_1, a_2) = (m \cdot a_1, m \cdot a_2))$. Since we're taking semantics in the category of $\mathcal{M}\text{-Act}$, we would expect $\text{Bool} \times \text{Bool}$ to be interpreted as

$$\begin{aligned} & U(F_{\mathcal{M}\text{-Act}}(\{\text{true}, \text{false}\}) \times F_{\mathcal{M}\text{-Act}}(\{\text{true}, \text{false}\})) \\ &= (\mathcal{M} \times \{\text{true}, \text{false}\}) \times (\mathcal{M} \times \{\text{true}, \text{false}\}) \end{aligned}$$

That is, this term doesn't print anything; it waits for a projection to be performed, at which it gives a Bool and produces a string. This is a *lazy semantics* (or call-by-name), as opposed to the *eager semantics* (or call-by-value) above.

Note that the definition of products in $\mathcal{M}\text{-Act}$ gives us the lazy semantics for beeps and boops: $\text{beep}; (M_1, M_2) = (\text{beep}; M_1, \text{beep}; M_2)$. This would not be the case in an eager language, where we would expect the RHS to beep twice.

2.1 Unit type

What about the unit type: $\cdot \vdash M : 1$? Again, we have two kinds of semantics:

- eager: $F_{\mathcal{M}\text{-Act}}1$, i.e. a series of beeps and boops and trivial value.
- lazy: $1_{\mathcal{M}\text{-Act}}$, i.e. nothing. This semantics validates the η rule for unit types, and acts more like a pure language; terms of unit type are just dead code.

2.2 Function types

What about function types: $\cdot \vdash M : \text{Bool} \Rightarrow \text{Bool}$?

- eager: $F_{\mathcal{M}\text{-Act}}(\text{Bool} \rightarrow UF_{\mathcal{M}\text{-Act}}\text{Bool})$, i.e. a series of beeps and boops, and a function that, given a boolean, will produce more beeps and boops and another boolean.
- lazy: $UF_{\mathcal{M}\text{-Act}}\text{Bool} \rightarrow F_{\mathcal{M}\text{-Act}}\text{Bool}$. The input is not evaluated: $(\lambda x. M) (\text{beep}; N) = M[\text{beep}; N/x]$. Similar to the unit type case, this validates the η rule for functions.

In the lazy case, we use the following notion of functions that are not necessarily equivariant. It turns out that this construction has the universal property in $\mathcal{M}\text{-Act}$ (e.g. $\mathcal{M}\text{-Act}(A', X \rightarrow A) \cong \mathcal{M}\text{-Act}(A' \times F_{\mathcal{M}\text{-Act}}X, A)$).

$$\frac{X \text{ set} \quad A \mathcal{M}\text{-Act}}{X \rightarrow A \mathcal{M}\text{-Act} := (|A|^X, (m \cdot f)(x) = m \cdot f(x))}$$

We see that interpreting the semantics of beep-boop requires universal properties in both Set and $\mathcal{M}\text{-Act}$. This is a general pattern when modelling effects.

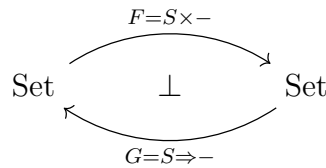
3 Mutable state

Let's now consider a language with a single mutable variable of type S . The behaviour of a program depends on the initial state of the mutable variable and will produce, as a side effect, the final state.

$$\cdot \vdash M : \text{Bool} \rightsquigarrow S \rightarrow S \times \text{Bool}$$

But instead of manually considering what the interpretation of what unit, products, functions, etc. are, we can use a common adjunction.

Is there an adjunction $F \dashv G$ such that $S \rightarrow S \times \text{Bool}$ is of the form $GF\{\text{true}, \text{false}\}$? Yes, and we previously saw that this was an adjunction for any cartesian closed category.



Then, we get the following interpretations, in which this adjunction naturally produces the right semantics for both eager and lazy languages:

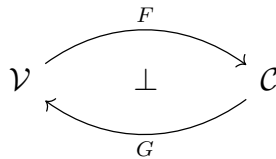
| | type | eager | lazy |
|--|---------------------------------------|---|---|
| | 1 | $S \rightarrow S \times 1$ | $S \rightarrow 1$ |
| | $\text{Bool} \times \text{Bool}$ | $S \rightarrow S \times (\text{Bool} \times \text{Bool})$ | $S \rightarrow (S \times \text{Bool}) \times (S \times \text{Bool})$ |
| | $\text{Bool} \Rightarrow \text{Bool}$ | $S \rightarrow S \times (\text{Bool} \rightarrow S \times \text{Bool})$ | $S \rightarrow (S \rightarrow S \times \text{Bool}) \rightarrow S \times \text{Bool}$ |

4 Call-by-push-value

Call-by-push value (CBPV) is a type theory to reason about effectful programs.

- CBPV subsumes both CBV and CBN: we can translate both into CBPV in a way that preserves the equational theory.
- We get sound and complete semantics in an adjunction between “nice” categories.

In particular, we have a category \mathcal{V} of *pure values* (often Set) that is very much like a model of STT (e.g. bicartesian closed) and a category \mathcal{C} of *effectful computations* which is much like a model of UnTT (e.g. bicartesian). In this way, we're kind of combining together two types theories that we've already seen.



4.1 Basic rules

We have 2 kinds of types:

- value types A_1, A_2, \dots
- computation types $\underline{B}_1, \underline{B}_2, \dots$

“Values” (or “pure functions”), written V , are governed by the judgment $\Gamma \vdash V : A$, where Γ is a context of value types $x : A, \dots$

“Effectful computations” (or “strict/linear homomorphisms”) are governed by the judgment $\Gamma \mid \underline{\Delta} \vdash M : \underline{B}$, where Γ is the same context of value types. The *stoup* may be *empty* (no variable) or *full* (single linear variable): $\underline{\Delta} ::= \cdot \mid \bullet : \underline{B}$.

Then, the denotation of these terms are morphisms in \mathcal{V} and \mathcal{C} :

| term | denotation |
|--|---|
| $\Gamma \vdash V : A$ | $\llbracket V \rrbracket : \mathcal{V}(\times \llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$ |
| $\Gamma \mid \cdot \vdash M : \underline{B}$ | $\llbracket M \rrbracket : \mathcal{C}(F \llbracket \Gamma \rrbracket, \llbracket B \rrbracket)$ |
| $\Gamma \mid \bullet : \underline{B}_1 \vdash M : \underline{B}_2$ | $\llbracket M \rrbracket : \mathcal{C}(F \llbracket \Gamma \rrbracket \times \llbracket B_1 \rrbracket, \llbracket B_2 \rrbracket)$ |

Note that \mathcal{C} does not necessarily generally have products, but it should have products with $F \llbracket G \rrbracket$. It’s also not the case in general that this is the Cartesian product; instead it is more like a tensor product.