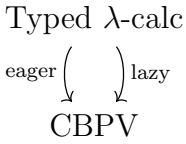
Lecture 23: Call-By-Push-Value II

Lecturer: Max S. New
Scribe: Elanor Tang

April 5, 2023

To recap the main idea behind CBPV calculus, we want to take STT terms minus the equations (also called typed λ-calculus) and have translations into CBPV that correspond to either eager and lazy evaluation. The point is to impose equality in CBPV (a kind of intermediate language) rather than in STT.



CBPV is similar to many other calculi:

- CPS - continuation passing style
- ANF - A-normal form
- SSA - static single assignment
- Monadic normal form

We choose to talk about CBPV since it is the nicest version from a type-theoretic perspective. Here in CBPV we have equations even in the presence of evaluation order, including η equations—the lack of η equations in effectful languages is reflected in the translations.

1 CBPV Types and Terms

CBPV has two kinds of types: value types A and computation (abbreviated comp) types $\underline{B}$. We denote this as

$$A \text{ vtype} \qquad\qquad \underline{B} \text{ ctype}$$

There are two kinds of corresponding terms

$$\Gamma \vdash V : A \qquad\qquad \Gamma \mid \underline{\Delta} \vdash M : \underline{B}$$

where Γ is a context of value types and $\underline{\Delta}$ is a context of comp types which is either empty or has a single variable $\bullet$ of comp type. That is,

$$\underline{\Delta} ::= \cdot \mid \bullet : \underline{B}'$$

Here we can think of the value terms as being "pure" values or functions—these behave like STT terms. We think of the comp terms as "effectful" computations (when $\underline{\Delta}$ input is empty) or strict/linear functions of the input (when $\underline{\Delta}$ has a variable). This corresponds to the fact that in many models, terms with a single variable correspond to homomorphisms. Here, linearity means that we do the $\bullet$ computation first when evaluating M, and we perform this exactly once.

Altogether, this is what we call the judgmental structure of the type theory. In our model, this would correspond to our notion of a CT structure.

2 Substitution Principle

We define the rules for substitution. These are all admissible, which we will ensure with the term constructs. We denote admissibility with $*$.

First, we can substitute a value into a value. This rule is the same as STT substitution.

$$\frac{\gamma : \Gamma' \to \Gamma \qquad \Gamma \vdash V : A}{\Gamma \vdash V[\gamma] : A} *$$

We can also substitute a value into a computation/linear term.

$$\frac{\gamma : \Gamma' \to \Gamma \qquad \Gamma \mid \underline{\Delta} \vdash M : \underline{B}}{\Gamma' \mid \underline{\Delta} \vdash M[\gamma] : \underline{B}} *$$

Lastly, we can substitute a computation into a computation.

$$\frac{\Gamma \mid \underline{\Delta} \vdash M : \underline{B} \qquad \Gamma \mid \bullet : \underline{B} \vdash N : \underline{B}'}{\Gamma \mid \underline{\Delta} \vdash N[M] : \underline{B}'} *$$

We write $N[M]$ instead of $N[M/\bullet]$ since we always substitute for $\bullet$. This is like plugging something into an evaluation context in operational semantics: to evaluate N, we will evaluate M first.

As before, these rules will have equations of associativity, etc. that we can prove are valid.

3 Identity/Variable Rules

These rules are also similar to STT rules. First, variables themselves stand for pure values and don't perform computations.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

We also have the comp variable, for any Γ.

$$\frac{}{\Gamma \mid \bullet : \underline{B} \vdash \bullet : \underline{B}}$$

This is a kind of trivial evaluation context. We'll talk about the equations for these rules later.

4 Connectives

4.1 Value Product Types

Value product types behave the same as before, except they are restricted to values. First we define the value product type:

$$\frac{A_1 \text{ vtype} \qquad A_2 \text{ vtype}}{A_1 \times A_2 \text{ vtype}}$$

with introduction rule

$$\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

and elimination rules for $i \in \{1, 2\}$.

$$\frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i V : A_i}$$

The β and η rules are also the same as before, so we will not repeat them here.

The rules for the unit type are as follows:

$$\frac{}{1 \text{ vtype}} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash V : 1}{\Gamma \vdash V = () : 1}$$

We say () is the only term of unit type. That is, the unit type carries trivial data.

4.2 Return Types

Given a value type A, we can define a corresponding comp type describing the programs which can produce side effects and in the end return values of type A.

$$\frac{A \text{ vtype}}{\text{Ret } A \text{ ctype}}$$

(Levy writes this rule as FA because it corresponds to the left adjoint in the semantics—we prefer "return" because it matches programming more closely.)

We define the introduction rule as follows:

$$\frac{\Gamma \vdash V : A}{\Gamma \mid \cdot \vdash \mathtt{ret}\, A : \mathrm{Ret}\, A}\ \textsc{RetI}$$

Here, the comp context is empty: when we're returning, we are not doing any further computations, so it does not use any input.

For the elimination rule, we have

$$\frac{\Gamma \mid \underline{\Delta} \vdash M : \mathrm{Ret}\, A \qquad \Gamma, x : A \mid \cdot \vdash N : \underline{B}}{\Gamma \mid \underline{\Delta} \vdash \mathtt{var}\ x = M; N : \underline{B}}\ \textsc{RetE}$$

Given a comp term N which knows how to use A values, we can execute M until it (possibly) produces a value which we bind to x, then we run N. Note that M might never produce a value if we allow effects such as program crashing. Thus, CBPV is a generic language for talking about effects: it has all kinds of adjunction models for different effects, and we add in domain specific rules for particular effects. We'll see an example of this later. (We can also choose to make our return rule be invisible syntactically, but here we describe it explicitly.)

We notice that this return type is also similar to coproducts: we could alternatively write it as

$$\mathrm{case}_{\mathrm{Ret}}\, M\{\mathtt{ret}\, x.N\}$$

As such, it has a similar contravariant universal property, and the β and η rules reflect that. In particular, the β rule tells us that we have the introduction form, we can simplify.

$$\frac{\Gamma \vdash V : A \qquad \Gamma, x : A \mid \cdot \vdash M : \underline{B}}{\Gamma \mid \cdot \vdash \mathtt{var}\ x = \mathtt{ret}\, V; M = M[V/x] : \underline{B}}\ \textsc{Ret}\beta$$

Here we use an empty context since the introduction rule has an empty context.

Note that this rule encompasses several compiler optimizations: if x is never used, then it represents dead code elimination. If x appears multiple times in M, this represents common subexpression elimination when viewed from right to left. Thus, we can justify compiler optimizations using the equational theory. Further, this demonstrates the necessity of static analysis to determine if a subexpression is a pure term, since this is a precondition to many compiler optimizations (such as the ones described above).

For the η rule, we have

$$\frac{\Gamma \mid \bullet : \mathrm{Ret}\, A \vdash M : \underline{B} \qquad \Gamma \mid \underline{\Delta} \vdash N : \mathrm{Ret}\, A}{\Gamma \mid \underline{\Delta} \vdash M[N] = (\mathtt{var}\ x = N; M[\mathtt{ret}\, x]) : \underline{B}}\ \textsc{Ret}\eta$$

Note that M is linear in N: the first thing it should do is whatever N does, which is represented by this rule. With this rule, we also finally include our semi-colon, meaning we have a real PL. :)

4.3 Sum Types

When restricted to pure values, the sum types behave the same as before; with computations, it is slightly different. We first introduce our new types:

$$\frac{A_1 \text{ vtype} \qquad A_2 \text{ vtype}}{A_1 + A_2 \text{ vtype}} \qquad\qquad \frac{}{0 \text{ vtype}}$$

with the same introduction rules from STT, restricted to values.

$$\frac{\Gamma \vdash V : A_1}{\Gamma \vdash i_1 V : A_1 + A_2} \qquad\qquad \frac{\Gamma \vdash V : A_2}{\Gamma \vdash i_2 V : A_1 + A_2}$$

The elimination type on values is also the same:

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash V_1 : A' \qquad \Gamma, x_2 : A_2 \vdash V_2 : A'}{\Gamma \vdash \text{case}_+ V \begin{cases} i_1 x_1 \to V_1 \\ i_2 x_2 \to V_2 \end{cases} : A'}$$

Note that the case analysis is on pure expressions with no side effects. But it would be inconvenient if we could only do a case analysis where the branches are pure values—thus, we also allow elimination into comp terms.

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \mid \underline{\Delta} \vdash M_1 : \underline{B} \qquad \Gamma, x_2 : A_2 \mid \underline{\Delta} \vdash M_2 : \underline{B}}{\Gamma \mid \underline{\Delta} \vdash \text{case}_+ V \begin{cases} i_1 x_1 \to M_1 \\ i_2 x_2 \to M_2 \end{cases} : \underline{B}}$$

Abusing notation slightly, we write the β and η rules for value and comp terms simultaneously, where T represents a value or a computation. For $j \in \{1, 2\}$, we have β rules

$$\text{case}_+ i_j V \begin{cases} i_1 x_1 \to T_1 \\ i_2 x_2 \to T_2 \end{cases} \overset{\beta+}{=} T_j[V/x_j]$$

and for $V : A_1 + A_2$, we have η rule

$$T[V/x] \overset{\eta+}{=} \text{case}_+ i_j V \begin{cases} i_1 x_1 \to T_1[i_1 x_1/x] \\ i_2 x_2 \to T_2[i_2 x_2/x] \end{cases}$$

Note that this is only valid since V is pure: if it produced side effects, we might get a different result when we lift it out of T.

For the empty type, we have elimination rules

$$\frac{\Gamma \vdash V : 0}{\Gamma \vdash \text{case}_0 V\{\} : A} \qquad\qquad \frac{\Gamma \vdash V : 0}{\Gamma \mid \underline{\Delta} \vdash \text{case}_0 V\{\} : \underline{B}}$$

Here this comp context can be arbitrary, since if we have proven 0, the context is inconsistent and the entire program is dead code anyway.

For $V : 0$, the η rule is given by

$$T[V/x] \overset{\eta 0}{=} \text{case}_0 V\{\}$$

As before, if V were under a variable binding, this rule would not make sense.

4.4 Computation Product Types

We can think of the value product types defined before as tuples, whereas the computation product types are more like lazy products or "objects," in the sense that projections are viewed as methods of the product, and we never evaluate a product unless we choose the projection first. To introduce the new comp product type, we use & to denote the difference from the value product type.

$$\frac{\underline{B}_1 \text{ ctype} \qquad \underline{B}_2 \text{ ctype}}{\underline{B}_1 \,\&\, \underline{B}_2 \text{ ctype}}$$

In words, we might say this type is "$\underline{B}_1$ with $\underline{B}_2$."

For the elimination rules for $i \in \{1, 2\}$, we have

$$\frac{\Gamma \mid \underline{\Delta} \vdash M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \mid \underline{\Delta} \vdash M.\pi_i() : \underline{B}_i}$$

The arbitrary context $\underline{\Delta}$ means we are allowed to have a comp variable. We also see that projection is strict on the input: if M crashes, $M.\pi_i()$ will also crash. We write $M.\pi_i()$ instead of $\pi_i M$ to emphasize how this projection behaves as a method call rather than a tuple projection.

For the introduction rule, we have

$$\frac{\Gamma \mid \underline{\Delta} \vdash M_1 : \underline{B}_1 \qquad \Gamma \mid \underline{\Delta} \vdash M_2 : \underline{B}_2}{\Gamma \mid \underline{\Delta} \vdash (M_1, M_2) : \underline{B}_1 \,\&\, \underline{B}_2}$$

At first glance, this could appear to violate linearity, since we are supposed to evaluate the term in the hole first, but we would move the hole into both sides of the product. Here, however, it is okay since we use lazy evaluation: we never evaluate this term unless we first project out one of the sides. Thus linearity is preserved.

We can also view this comp product introduction as akin to the λ case:

$$\lambda \begin{cases} .\pi_1() \to M_1 \\ .\pi_2() \to M_2 \end{cases}$$

This is called copattern matching. In some ways, comp products are similar to sum type comp elimimation, except that here the two cases can have different types, whereas for sums the two cases had to have the same type.

Using this notation, we define the β and η rules:

$$\left(\lambda \begin{cases} .\pi_1() \to M_1 \\ .\pi_2() \to M_2 \end{cases}\right).\pi_i() \overset{\beta\&}{=} M_i$$

$$\frac{\Gamma \mid \underline{\Delta} \vdash M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \mid \underline{\Delta} \vdash M = \lambda \begin{cases} .\pi_1() \to M_1.\pi_1() \\ .\pi_2() \to M_2.\pi_2() \end{cases} : \underline{B}_1 \,\&\, \underline{B}_2} \,\eta\&$$

The β rule demonstrates how $M : \underline{B}_1 \,\&\, \underline{B}_2$ is evaluated lazily, as stated before. In the η rule, we can have an arbitrary context here since there is an arbitrary context in the comp product introduction.

4.5 Function Types

When it comes to function types, there are three possibilities to consider:

1. $A \to \underline{B}$ ctype, which are computations that take a value of type A and do something.
2. $A \Rightarrow A'$ vtype, which are pure functions from A to A'.
3. $\underline{B} \multimap \underline{B}'$ vtype, which are linear functions from $\underline{B}$ to $\underline{B}'$.

We will go with the first option, since this most closely corresponds to functions in effectful PLs. Then we have function type rule

$$\frac{A \text{ vtype} \qquad \underline{B} \text{ ctype}}{A \to \underline{B} \text{ ctype}}$$

with introduction

$$\frac{\Gamma, x : A \mid \underline{\Delta} \vdash M : \underline{B}}{\Gamma \mid \underline{\Delta} \vdash \lambda x.M : A \to \underline{B}}$$

and application elimination

$$\frac{\Gamma \mid \underline{\Delta} \vdash M : A \to \underline{B} \qquad \Gamma \vdash V : A}{\Gamma \mid \underline{\Delta} \vdash M\,V : \underline{B}}$$

where application is linear in M, since we use an arbitrary comp context.

The β and η rules are as follows:

$$(\lambda x.M)\,V \stackrel{\beta}{=} M[V/x]$$

$$M : A \to \underline{B} \stackrel{\eta}{=} \lambda x.M\,x$$

The η rule reflects how this is a lazy function type: we wait until we're given input to evaluate the function. This rule would not hold under eager evaluation, since on the left side M might execute and produce side effects.

4.6 Closures

The return type gave us a way to get a comp type from a value type; now closures give a way to get a value type from a comp type. The type rule is

$$\frac{\underline{B} \text{ ctype}}{\text{Closure}\,\underline{B} \text{ vtype}}$$

We can also write Thunk $\underline{B}$ instead of Closure $\underline{B}$. This type represents the values which are suspended computations: we don't execute them unless we call them. This allows us to write higher order programs which pass around computations as values.

For the introduction rule, we have

$$\frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \vdash \texttt{proc}\,\{M\} : \text{Closure}\,\underline{B}}$$

where "proc" stands for "procedure," since we suspend M as a procedure that can be executed. Here M can use any variables in the current context Γ, so we need that context for the closure of M to typecheck.

The elimination rule is

$$\frac{\Gamma \vdash V : \text{Closure}\,\underline{B}}{\Gamma \mid \cdot \vdash V.\texttt{call()} : \underline{B}}$$

Here we have an empty context, since there is no input when we call the procedure.

The β and η rules enforce that introduction and elimination are inverses of each other. For the β rule, we have

$$(\texttt{proc}\,\{M\}).\texttt{call()} \stackrel{\beta}{=} M$$

which validates code inlining. For the η rule, given that $V : \text{Closure}\,\underline{B}$, we have

$$\texttt{proc}\,\{V.\texttt{call()}\} \stackrel{\eta}{=} V$$

which means that you can't observe the number of procedure calls. This rule is necessary for most compiler optimizations.

5 Example: Beep and Boop

We now return to our motivating example of printing beeps and boops to demonstrate how we can add effects with additional rules. In particular, we add rules

$$\frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \mid \cdot \vdash \texttt{beep}; M : \underline{B}} \qquad \frac{\Gamma \mid \cdot \vdash M : \underline{B}}{\Gamma \mid \cdot \vdash \texttt{boop}; M : \underline{B}}$$

Here we have an empty context since we want `beep` or `boop` to be the first thing we execute, rather than whatever is in the comp context. We also note that if we have the return type, we could derive the above rules from

$$\frac{}{\Gamma \mid \cdot \vdash \texttt{beep} : \text{Ret}\,1} \qquad \frac{}{\Gamma \mid \cdot \vdash \texttt{boop} : \text{Ret}\,1}$$

but here we will add beeps and boops explicitly.

Since we care about the uniqueness of beep and boop printing, we do not want beep and boop to commute. We do, however, want a general rule saying that every computation with a free comp variable should commute with these new effects.

$$\frac{\Gamma \mid \bullet : \underline{B} \vdash M : \underline{B}' \qquad \Gamma \mid \cdot \vdash N : \underline{B}}{\Gamma \mid \cdot \vdash M[\texttt{beep}; N] = \texttt{beep}; M[N] : \underline{B}'}$$

This rule enforces strictness, or in other words, it says that substitution is equivariant. Note that since we allow the $\bullet$ variable, strictness applies to many of the previous rules. For example,

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

since the rule governing return type elimination allows for a $\bullet$ context.