Lecture 24

Lecturer: Max S. New
Scribe: Matt Wang

April 10, 2023

Today we'll talk more about effects and CBPV.

1 Review

Last time we saw how to add beeps and boops to CBPV with the two rules:

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

We also added the following strictness rule

$$\frac{\Gamma \mid \cdot \vdash M : B_1 \qquad \Gamma \mid B_1 \vdash N : B}{\Gamma \mid \cdot \vdash N[\texttt{beep}; M] = \texttt{beep}; N[M] : B}$$

and similarly for boop. For other effects besides beeps and boops we are going to add additional equations that help us reason about our programs and tell us something more about how the compiler has to be implemented.

One equation for beep and boop we can add is $\texttt{beep}; \texttt{boop}; M = \texttt{boop}; \texttt{beep}; M$. This equation tells us any sequence of beeps/boops can be reordered arbitrarily. Without this commutativity rule, the canonicity theorem we wanted was that any term M was equal to some sequence of beeps and boops with unique ordering. With this new rule only the total count of beeps and boops matters.

Similarly, if we were to add the not very-useful rule $\texttt{beep}; M = \texttt{boop}; M$ then we only care about the total number of operations. We could also add $\texttt{beep}; \texttt{beep}; M = \texttt{beep}; M$ we only care if we beep before the next boop. Whatever equation we add to our language depends on the purpose of the program. Now we'll consider some effects with more natural equations.

2 Mutable State

To keep it simple, say we have one global boolean memory location that we can read from or update. Without assuming we have any particular connectives, we can make the following rules to set state:

$$\frac{\Gamma \mid \cdot \vdash M : B}{\Gamma \mid \cdot \vdash \texttt{set }\ 0; M : B} \qquad \frac{\Gamma \mid \cdot \vdash M : B}{\Gamma \mid \cdot \vdash \texttt{set }\ 1; M : B}$$

To read from state without introducing a boolean connective, we'll perform a case split on the current state of the memory location instead of giving a single read operation that returns a boolean:

$$\frac{\Gamma \mid \cdot \vdash M_0 : B \qquad \Gamma \mid \cdot \vdash M_1 : B}{\Gamma \mid \cdot \vdash \texttt{if}_{read}\, M_0\, M_1 : B}$$

Note: $\texttt{if}_{read}$ requires an empty context.

If this is to act like mutable state, then we want to add in equations that internalize the intuitive semantics. First we'll add in the following linearity rules:

Any computation substituting in a set of a bit commutes with any computation with substitution into the computation:

$$N[\texttt{set }\ b; M] = \texttt{set }\ b; N[M]$$

For $\texttt{if}_{read}$, if we substitute a read statement into a program commutes with reading the bit first:

$$N[\texttt{if}_{read} M_1 M_2] = \texttt{if}_{read}(N[M_1])(N[M_2])$$

We have the following intuitive laws:

- put; get
$$\texttt{set }\ i; \texttt{if}_{read} M_0\, M_1 = M_i$$
- put; put
$$\texttt{set }\ i; \texttt{set }\ i'; M = \texttt{set }\ i'; M$$
- get; get
$$\texttt{if}_{read}(\texttt{if}_{read} M_0\, M_1)(\texttt{if}_{read} M_2\, M_3) = \texttt{if}_{read} M_0\, M_3$$

  More generally:

$$\texttt{if}_{read}(\texttt{if}_{read} M_0\, M_1)\, M_2 = \texttt{if}_{read} M_0\, M_2$$

$$\texttt{if}_{read} M_0\, (\texttt{if}_{read} M_1\, M_2) = \texttt{if}_{read} M_0\, M_2$$

Note: if our language had concurrency then these laws wouldn't hold, e.g. the put; get law tells us that setting a bit and then reading is an atomic operation. These laws also wouldn't be valid if we could explicitly observe whenever the state is accessed, e.g. running a profiler on the program. When we want a particular equation to hold, it's up to us to determine whether things are actually observable or if we care.

3 Non-determinism/Backtracking

We can add non-determinism to our language by adding a fail and ambiguous operation:

$$\frac{}{\Gamma \mid \cdot \vdash \texttt{fail} \ : B} \qquad\qquad \frac{\Gamma \mid \cdot \vdash M_0 : B \qquad \Gamma \mid \cdot \vdash M_1 : B}{\Gamma \mid \cdot \vdash \texttt{amb} \ \ M_0\, M_1 : B}$$

Other than re-naming, so far the `amb` operation is the same as `if`. We give them different names because they differ in the equations we impose. Some natural equations for reasoning about non-determinism would be

- Associativity: $\texttt{amb} \ (\texttt{amb} \ \ M_0\, M_1)\, M_2 = \texttt{amb} \ \ M_0(\texttt{amb} \ \ M_1\, M_2)$
- Left cancel: $\texttt{amb fail} \ \ M_1 = M_1$
- Right cancel: $\texttt{amb} \ \ M_0\,\texttt{fail} \ \ = M_0$
- Idempotency: $\texttt{amb} \ \ M\, M = M$
- Fairness: $\texttt{amb} \ \ M_0\, M_1 = \texttt{amb} \ \ M_1\, M_0$

These are exactly the rules for the algebraic theory of semi-lattices, which are equivalent to idempotent commutative monoids. The canonicity result for this calculus is that under these equations, we can prove that any term

$$\Gamma \mid \cdot \vdash M : \mathrm{RetBool}$$

is an element of the free semi-lattice on Booleans which is just the finite power set, i.e. $M = \lfloor S \rfloor \in \mathcal{P}(\mathrm{Bool})$. So M could fail ($\emptyset$), return true ($\{1\}$), return false ($\{0\}$, or be ambiguous ($\{0,1\}$. Most programming languages do not support all of these equations, e.g. Prolog is not unbiased because you can observe the ordering of choices taken.

4 Writing programs in CBPV

We can explain the semantics of STT terms extended with beeps and boops by translating them to CBPV.

$$(\texttt{beep}; (\lambda x.(\texttt{boop}; x+1)))(\texttt{beep}; 5)$$

is a term in STT with numbers. It could evaluate to any of the following depending on evaluation order:

- $\texttt{boop}; \texttt{beep}; \texttt{boop}; 6$ (call by value/eager, left-to-right)
- $\texttt{boop}; \texttt{boop}; \texttt{beep}; 6$ (call by name)

- `beep`; `boop`; `boop`; 6 (call by value/eager, right-to-left)

There is some ambiguity in the source language syntax, not clear how it will evaluate without knowing more about the language. We can reason about all of these evaluation orders by different translations of the λ term into CBPV in a way that encodes the evaluation oder. Informally, we'll perform a translation that is syntactic translation from arbitrary λ terms into CBPV.

First of all, our program is of type Int in STT but it is an effectful computation that produces an int so it has type RetInt in CBPV. More generally, we have the following translations from STT:

$$\begin{aligned} M : \text{Int} \rightsquigarrow \lceil M \rceil &: \text{RetInt} \\ M : \text{Int} \implies \text{Int} \rightsquigarrow \lceil M \rceil^{CbV} &: \text{Ret}\,(\text{Closure}\,(\text{Int} \to \text{RetInt})) \\ M : \text{Int} \implies \text{Int} \rightsquigarrow \lceil M \rceil^{CbN} &: \text{Closure}\,(\text{RetInt}) \to \text{RetInt} \end{aligned}$$

The generalization of these translations to an arbitrary M of type A in CBPV is

$$\begin{aligned} M : A \rightsquigarrow \lceil M \rceil^{CbV} &: \text{Ret}\,\lceil A \rceil^{CbV} \\ M : B \rightsquigarrow \lceil M \rceil^{CbN} &: \lceil B \rceil^{CbN} \end{aligned}$$

and variables are directly translated in CbV but in CbN they are translated to a closure:

$$\begin{aligned} x_0 : A_0 \vdash M : A \rightsquigarrow x_0 : \lceil A_0 \rceil^{CbV} \vdash \lceil M \rceil^{CbV} : \text{Ret}\,\lceil A \rceil^{CbV} \\ x_0 : B_0 \vdash M : B \rightsquigarrow x_0 : \text{Closure}\,\lceil B_0 \rceil^{CbN} \vdash \lceil M \rceil^{CbN} : \lceil B \rceil^{CbN} \end{aligned}$$

From this perspective we see why some of the η-rules are valid in CbV but not CbN and vice versa. The η-rules for function and product types are about terms of function and product types and in the translation we translate to CbN terms with a similar connective. For things like sum types, in CbV the variables are guaranteed to be actual values so we have stronger η-rules.

To translate functions from STT types to CbV we have

$$\lceil C_1 \to C_2 \rceil^{CbV} := \text{Closure}\,(\lceil C_1 \rceil^{CbV} \to \text{Ret}\,\lceil C_2 \rceil^{CbV})$$

and in CbN we have

$$\lceil C_1 \to C_2 \rceil^{CbN} := \text{Closure}\,\lceil C_1 \rceil^{CbN} \to \lceil C_2 \rceil^{CbN}$$

To translate function application MN in CbN we note that M is a function with type above, taking a closure of type C_1 and outputting C_2 and we'll pass it an unevaluated version of N:

$$\lceil MN \rceil^{CbN} := \lceil M \rceil^{CbN}(\texttt{proc}\,\{\lceil N \rceil^{CbN}\})$$

and in CbV with left to right evaluation we evaluate M to a value and N to a value and then apply M to N:

$$\lceil MN \rceil^{CbV,ltr} := \mathrm{var}\, f = \lceil M \rceil^{CbV,ltr}; \mathrm{var}\, x = \lceil N \rceil^{CbV,lt}; f.\texttt{call()}\,x$$

To get right to left evaluation we would just swap the order of evaluations of M and N.

To translate sum types in CbV we can just use the sum types for value types:

$$\lceil C_1 + C_2 \rceil^{CbV} := \lceil C_1 \rceil^{CbV} + \lceil C_2 \rceil^{CbV}$$

but in CbN our output needs to be a computation type so we will translate it to something that returns a value of a sum type:

$$\lceil C_1 + C_2 \rceil^{CbN} := \mathrm{Ret}\,(\mathrm{Closure}\,\lceil C_1 \rceil^{CbN} + \mathrm{Closure}\,\lceil C_2 \rceil^{CbN}).$$

To translate case statements into CbV we first need to evaluate M to a value:

$$\lceil \mathrm{case}_{+} M \begin{cases} i_0 x_0 \to N_0 \\ i_1 x_1 \to N_1 \end{cases} \rceil^{CbV} := \mathrm{var}\, s = \lceil M \rceil^{CbV}; \mathrm{case}_{+} s \begin{cases} i_0 x_0 \to \lceil N_0 \rceil^{CbV} \\ i_1 x_1 \to \lceil N_1 \rceil^{CbV} \end{cases}$$

and the call by name translation is the same but with call by name evaluation because the input is linearly used.