Scribe: Tom Roeder, Chavdar Botev Lecturer: Radu Rugina

1 Review

We started talking last time about a new language, called REC, which we use for exploring the semantics of functions. A program in REC consists of k functions, followed by an expression.

$$f_1(x_{11}, \ldots, x_{1a_1}) = e_{f_1}$$

$$f_k(x_{k1}, \ldots, x_{ka_k}) = e_{f_k}$$

The first function has a_1 arguments, the k-th function has a_k arguments, and the body of the function is an expression in the language. These functions may be recursive, i.e. you may have recursive function invocations in the body of these functions. The syntax for these expression doesn't include sequences of commands. On the other hand, we have arithmetic expressions in the language.

We model booleans using integer values. Let's make this clear, because there was confusion. The comparison $a \leq b$ gives an integer number, which is 1 if true, 0 if false. Also we have conditional expressions:

$$\textbf{ifnz}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1$$

Such an expression evaluates to e_0 if e evaluates to a non-zero, and takes the false branch otherwise. Note that this is opposite to the way Winskel does it.

2 Call-By-Value Semantics of REC

2.1 Operational Semantics

Last time, we started talking about semantics for our language. We will briefly review it here: there are two kinds of parameter passing. The first is Call-by-Value (CBV) (we discussed it): evaluate the arguments first, then pass the results on to the function. We gave the rules; we merely review here the function definition:

$$\frac{e_{i1} \to n_1, \ldots, e_{ia_i} \to n_{a_i}\ \ e_{f_i}[x_{i1} \to n_{i1}, \ldots, x_{ia_i} \to n_{ia_i}] \to n}{f_i(e_{i1}, \ldots, e_{ia_i}) \to n}$$

It's fairly explicit that we first evaluate the arguments, and then we evaluate the body of the function using the computed values for the arguments. The other rules are identical to those that we have stated before for REC, so let us move on to the Denotational Semantics of CBN.

2.2 Denotational Semantics

We said that we want to make a denotation which takes a variable environment and returns an integer value or bottom for non-termination: $\mathcal{C}[\![e]\!] : \mathbf{Env} \to \mathbb{Z}_\perp$. An environment is a mapping of $\mathbf{Env} : \mathbf{Var} \to \mathbb{Z}$. The semantics are as follows:

Constants use the obvious definition

$$[\![n]\!]\rho = \lfloor n \rfloor$$

For variables, we have

$$[\![x]\!]\rho = \lfloor \rho(x) \rfloor$$

For arithmetic, we must use operations for lifted integers

$$[\![e_0 \text{ op } e_1]\!]\rho = [\![e_0]\!]\rho\ \ \text{op}_\perp\ [\![e_1]\!]\rho$$

We also have

$$[\![\textbf{ifnz}\ e\ \textsf{then}\ e_0\ \textsf{else}\ e_1]\!]\rho = \textbf{condnz}([\![e]\!]\rho, [\![e_0]\!]\rho, [\![e_1]\!]\rho)$$

We thought that we could construct the **condnz**, like the **cond** that we had before: the disjoint sum of two functions and combining them. By the way, all these constructs are based on the simple constructs from our domain construction, so we know they are continuous.

For functions calls with CBV semantics, we evaluate each argument, and if the evaluation of any argument doesn't terminate, then the evaluation of the call doesn't terminate. We have

$$[\![f_i(e_{i1}, \ldots, e_{ia_i})]\!]\rho = \text{let } n_{i1} = [\![e_{i1}]\!]\rho, \ldots, n_{ia_i} = [\![e_{ia_i}]\!]\rho\ .\ [\![e_{f_i}[x_{i1} \to n_{i1}, \ldots, x_{ia_i} \to n_{ia_i}]]\!]\rho$$

We now move on to Call-by-Name.

3 Call-by-Name Semantics of REC

3.1 Operational Semantics

Let's look at the operational semantics. We don't evaluate the parameters, we just pass them to the body, which will evaluate them only when it needs them. We will skip all the rules except the rule for evaluation of function calls.

$$\frac{e_{f_i}[x_{i1} \to e_{i1}, \ldots, x_{ia_i} \to e_{ia_i}] \to n}{f_i(e_{i1}, \ldots, e_{ia_i}) \to n}$$

To evaluate the call, we evaluate body of the function, e_{f_i}, after replacing each parameter with the corresponding expressions in the call. We won't put the other rules here.

3.2 Denotational Semantics

What we need for denotational semantics is pretty similar to what we had before. The main difference is that we need to change the environment to allow variables to be mapped to bottom values: **Env** : **Var** $\to \mathbb{Z}_\perp$. The rules are very similar to the ones for CBV. The main difference is the rule for function calls.

$$\begin{aligned}
[\![n]\!]\rho &= \lfloor n \rfloor \\
[\![x]\!]\rho &= \rho(x) \\
[\![e_0 \text{ op } e_1]\!]\rho &= [\![e_0]\!]\rho \text{ op}_\perp\ [\![e_1]\!]\rho \\
[\![\textbf{ifnz}\ e\ \textsf{then}\ e_0\ \textsf{else}\ e_1]\!]\rho &= \textbf{condnz}([\![e]\!]\rho, [\![e_0]\!]\rho, [\![e_1]\!]\rho)
\end{aligned}$$

The reason I'm writing them all again is that they invoke each other. This means that we have two systems of rules, one for CBN and one for CBV, and the rules in each system inductively invoke the rules in that system. Ideally, we should use explicit subscripts, such as $[\![e]\!]_{cbv}$ and $[\![e]\!]_{cbn}$, to indicate which semantic system we are using.

The CBN denotational semantics for function calls is:

$$[\![f_i(e_{i1}, \ldots, e_{ia_i})]\!]\rho = [\![e_{f_i}[x_{i1} \to [\![e_{i1}]\!]\rho, \ldots, x_{ia_i} \to [\![e_{ia_i}]\!]\rho]]\!]\rho$$

Any expression may evaluate to $\perp$, so $\perp$ becomes a parameter to a function. Therefore, REC functions with CBN semantics are modeled as $\mathbb{Z}_\perp^{a_i} \to \mathbb{Z}_\perp$. In contrast, with CBV semantics functions are mappings $\mathbb{Z}^{a_i} \to \mathbb{Z}_\perp$, since you cannot pass bottom as an argument. Let's do an example, and see where the problems are if there are any.

4 An Example and Problems

Recall the example from last time:

$$\begin{aligned} f(x) &= f(x) + 1 \\ g(y) &= 2 \end{aligned}$$

And we wish to evaluate, under our different schemes,

$$g(f(3))$$

The execution of f with any argument always invokes f recursively, therefore it doesn't terminate. Now consider $g(f(3))$. Under CBV, we first try to evaluate $f(3)$, and it won't terminate. With CBN, we don't evaluate $f(3)$, but just pass it to procedure g. We thus expect that CBN terminates and CBV does not. Let's see if we can get these results with the semantics we put up. We will note that the subscripts on the semantic brackets denote here the parameter passing style that we are using.

First we will do CBN. I'm going to apply the definition to this program. The denotation is the denotation of the body of g with its argument replaced with the evaluation of $f(3)$.

$$\begin{aligned} [\![g(f(3))]\!]_{\text{cbn}}\rho &= [\![2[y \to [\![f(3)]\!]_{\text{cbn}}\rho]]\!]_{\text{cbn}}\rho \\ &= [\![2]\!]_{\text{cbn}} = \lfloor 2 \rfloor \end{aligned}$$

The expression representing the body of g has no occurrence of y, so the denotation of $g(f(3))$ with CBN semantics is 2. That was simple. What happens with CBV?

$$\begin{aligned} [\![g(f(3))]\!]_{\text{cbv}}\rho &= \text{let } n = [\![f(3)]\!]_{\text{cbv}}\rho \;.\; [\![2[y \to n]]\!]_{\text{cbv}}\rho \\ &= \text{let } n = [\![f(3)]\!]_{\text{cbv}}\rho \;.\; \lfloor 2 \rfloor \end{aligned}$$

So we need to find the following

$$\begin{aligned} [\![f(3)]\!]_{\text{cbv}}\rho &= \text{let } m = [\![3]\!]_{\text{cbv}}\rho \;.\; [\![(f(x)+1)[x \to m]]\!]_{\text{cbv}}\rho \\ &= \text{let } m = \lfloor 3 \rfloor \;.\; [\![f(m)+1]\!]_{\text{cbv}}\rho \\ &= [\![f(3)+1]\!]_{\text{cbv}}\rho \\ &= [\![f(3)]\!]_{\text{cbv}}\rho +_{\perp} \lfloor 1 \rfloor \end{aligned}$$

Hence, the evaluation of $f(3)$ doesn't terminate, and neither does the evaluation of $g(f(3))$.

5 Fixing the Denotational Semantics of REC

We have a big problem here: the denotational semantics we presented (i.e. the definition of $\mathcal{C}[\![.]\!]$) is a set of recursive equations rather than a definition. So, whenever you have a rule, it might invoke other rules and get back to an invocation of itself. This is a system of mutually recursive equations (the system as a whole, not just the rule for functions). Note that this applies to both CBN and CBV semantics. We will discuss only the case of CBN; the recursive system for CBV can be similarly solved.

To solve the mutually recursive equations, we go back to our favorite tool, the fixed point. We will express the solution of this system as the fixed point of a higher order function. It is not obvious at this point what the higher order function is.

We first define an function environment **Fenv**:

$$\textbf{Fenv} : (\mathbb{Z}_{\perp}^{a_i} \to \mathbb{Z}_{\perp}) \times \cdots \times (\mathbb{Z}_{\perp}^{a_i} \to \mathbb{Z}_{\perp})$$

An element $\phi \in \mathbf{Fenv}$ is a tuple $\phi = (\phi_1, \ldots, \phi_k)$. You should think of $(\phi_1, \ldots, \phi_k)$ as an approximation of the denotational semantics of the set of functions $\{f_1, \ldots, f_k\}$.

Starting with some approximation $\phi = (\phi_1, \ldots, \phi_k)$, we can derive a new approximation ϕ' for the functions $\{f_1, \ldots, f_k\}$, using the the known approximations $(\phi_1, \ldots, \phi_k)$ whenever we reach function calls. The computation of ϕ' from ϕ is therefore not recursive. We formalize the iterative computation of $(\phi_1, \ldots, \phi_k)$ as follows. We define a helper relation $[\![e]\!]_h$, which is defined on $\mathbf{Fenv}$ and returns what our denotational semantics returned before.

$$[\![e]\!]_h : \mathbf{Fenv} \to [\mathbf{Env} \to \mathbb{Z}_\perp]$$

That means that whenever we want to evaluate an expression e, we take φ, which binds functions to their approximations, and ρ, which binds variables to their values, and use them to evaluate e. Then $[\![e]\!]_h$ is formally defined as follows:

$$\begin{aligned}
[\![n]\!]_h \varphi\rho &= \lfloor n \rfloor \\
[\![x]\!]_h \varphi\rho &= \rho(x) \\
[\![e_0 \text{ op } e_1]\!]_h \varphi\rho &= [\![e_0]\!]_h \varphi\rho \text{ op}_\perp\ [\![e_1]\!]_h \varphi\rho \\
[\![\mathbf{ifnz}\ e \ \mathsf{then}\ e_0 \ \mathsf{else}\ e_1]\!]_h \varphi\rho &= \mathbf{condnz}([\![e]\!]_h \varphi\rho, [\![e_0]\!]_h \varphi\rho, [\![e_1]\!]_h \varphi\rho)
\end{aligned}$$

Note that we propagate φ, because we're going to use it for function calls. Let's see what happens for function calls. The problem was that a function call could occur in the body of a function. We need a known interpretation of what this function is. If we have φ, then we can just apply φ_i, since φ contains all the φ_i's.

$$[\![f_i(e_{i1}, \ldots, e_{ia_i})]\!]_h \varphi\rho = \varphi_i([\![e_{i1}]\!]_h \varphi\rho, \ldots, [\![e_{ia_i}]\!]_h \varphi\rho)$$

Note that the above relations represent a definition of $[\![e]\!]_h$, not a set of recursive equations. The only thing left to do is to use this helper relation to compute a function environment. We can use it to build a higher-order function $F : \mathbf{Fenv} \to \mathbf{Fenv}$ as follows:

$$F(\varphi_1, \ldots, \varphi_k) \to (\varphi'_1, \ldots, \varphi'_k)$$

$$\begin{aligned}
F(\varphi_1, \ldots, \varphi_k) \quad = \quad & ([\![e_{f_1}[x_{11} \to n_{11}, \ldots, x_{1a_1} \to n_{1a_1}]]\!]_h \varphi\rho, \\
& \ldots \\
& [\![e_{f_k}[x_{k1} \to n_{k1}, \ldots, x_{ka_k} \to n_{ka_k}]]\!]_h \varphi\rho)
\end{aligned}$$

Now this F is a higher-order function, and it is straightforward to see that it is continuous, because we built it using known continuous functions. This means that it has a fixed point, and we can get to some final fixed point value φ^{fix}.

$$\text{fix}(F) = \varphi^{\text{fix}} = (\varphi_1^{\text{fix}}, \ldots, \varphi_k^{\text{fix}})$$

We can then construct our denotational semantics for the expressions as follows:

$$[\![e]\!]\rho = [\![e]\!]_h \varphi^{\text{fix}} \rho$$