cs611 Lecture 25 **Denotational semantics of F** 28 October 2002

Scribe: Kamal Aboul-Hosn and Ganesh Ramanarayanan Lecturer: Radu Rugina

Recall from before the definition of the language F:

$$\begin{array}{rcl} e & ::= & n \mid \text{true} \mid \text{false} \mid \\ & & e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_0 \mid e_0 \leq e_1 \mid e_0 = e_1 \mid \\ & & \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid x \mid \lambda x.\, e \mid e_0 e_1 \mid \mathsf{rec}\ y\ (\lambda x.\ e) \mid \\ & & \langle e_0, e_1 \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \end{array}$$

1 Denotational Semantics of F

We want to define denotational semantics for F with call-by-value semantics, left-to-right evaluation, and error handling.

1.1 Domain Equations

First, we will explicitly define several *semantic domains* relevant to our language. These domains define the set of values that certain expressions expect. For instance, the condition of an if expression should be in the domain $\mathbb{T}$ of truth values; and the argument of fst or snd should be in the domain *Pair* of pairs. Note that the relations below are recursive, therefore they are *recursive domain equations* (rather than definitions):

$$\begin{array}{rcl} v \in \mathit{Value} & = & \mathbb{Z} + \mathbb{T} + \mathit{Pair} + \mathit{Function} \\ n \in \mathbb{Z} & = & \{\ldots, -1, 0, 1, \ldots\} \\ t \in \mathbb{T} & = & \{\mathit{true}, \mathit{false}\} \\ p \in \mathit{Pair} & = & \mathit{Value} \times \mathit{Value} \\ f \in \mathit{Function} & = & \mathit{Value} \to \mathit{Result} \\ r \in \mathit{Result} & = & (\mathit{Value} + \mathit{Error})_\perp \\ \mathit{Error} & = & \{\mathit{error}\} \\ \rho \in \mathit{Env} & : & \mathit{Var} \to \mathit{Result} \end{array}$$

The denotation of an expression e evaluates e in an environment ρ and produces a result, which may be either a value, an error, or a non-terminating computation (represented by $\perp$):

$$\begin{array}{rcl} \mathcal{C}[\![e]\!] & \in & \mathit{Env} \to \mathit{Result} \\ \mathcal{C}[\![e]\!]\rho & \in & \mathit{Result} \end{array}$$

1.2 Case statements

When writing the denotational semantics of F, we will find it helpful to use *case* constructs to match values in sum domains (such as *Value*) to values in the component domains of the sum (such as $\mathbb{T}$, $\mathbb{Z}$, *Pair*, etc). Case constructs will allow us to concisely express this matching of domains when writing the denotational semantics.

A *case* construct has the following form:

$$\begin{array}{lcl} \mathit{case}\ \ e\ \ \mathit{of} & & \mathit{in}_1(x_1) \Rightarrow e_1 \\ & \mid & \mathit{in}_2(x_2) \Rightarrow e_2 \\ & & \vdots \\ & \mid & \mathit{in}_k(x_k) \Rightarrow e_k \end{array}$$

Recall disjoint sum domains (from the Domain Constructions lecture) and the injection functions for such domains. Each injection function in_k maps values from the component domain D_k to the disjoint sum domain:

$$in_k : D_k \to (D_1 + \cdots + D_k)_\perp$$

The *case* construct is just syntactic sugar for the expression that combines the functions $\lambda x_k.\, e_k$:

$$\begin{array}{rcl} case & : & (D_1 + D_2 + \cdots + D_k)_\perp \to D \\ case & \equiv & [\lambda x_1.\, e_1, \ldots, \lambda x_k.\, e_k]^* \\ \lambda x_i.\, e_i & \in & D_i \to D, \quad 1 \le i \le k \end{array}$$

The case construct takes an expression e as argument. It evaluates e to a value in $(D_1 + D_2 + \cdots + D_k)_\perp$. It e is $\perp$, the construct returns $\perp$ (because of the lifting operator $[\cdot]^*$); otherwise, it matches e with one of the component domains, and applies the corresponding function $\lambda x_k.\, e_k$.

We next extend the case construct with an else clause, which matches all of the non-listed cases. The case construct with an else clause has the following form (where $j < k$):

$$\begin{array}{lll} case\ \ e & of & in_1(x_1) \Rightarrow e_1 \\ & & \vdots \\ & \mid & in_j(x_j) \Rightarrow e_j \\ & \mid & else\ e' \end{array}$$

We can desugar the case construct with else clause into the following expression:

$$\begin{array}{rcl} case & \equiv & [\lambda x_1.\, e_1, \ldots, \lambda x_j.\, e_j, \lambda x.\, e']^* \\ \lambda x_i.\, e_i & \in & D_i \to D, \quad 1 \le i \le j \\ \lambda x.\, e' & \in & (D_{j+1} + \ldots + D_k) \to D \end{array}$$

Finally, we introduce the *ecase* construct, which doesn't have an else clause, but rather returns *error* for all of the unmatched cases:

$$\begin{array}{lll} ecase\ \ e & of & in_1(x_1) \Rightarrow e_1 \\ & & \vdots \\ & \mid & in_j(x_j) \Rightarrow e_j \end{array}$$

We assume that *error* is an element of the range domain: $error \in D$. The *ecase* construct is equivalent to:

$$\begin{array}{rcl} ecase & \equiv & [\lambda x_1.\, e_1, \ldots, \lambda x_j.\, e_j, \lambda x.\, error]^* \\ \lambda x_i.\, e_i & \in & D_i \to D, \quad 1 \le i \le j < k \\ \lambda x.\, error & \in & (D_{j+1} + \ldots + D_k) \to D \end{array}$$

To summarize, the *ecase* construct does the following:

- it returns $\perp$ when the argument expression evaluates to $\perp$;
- it returns *error* if e evaluates to a value that doesn't match any of the listed cases;
- if the evaluation of e yields a value in the domain D_k, it applies the abstraction $\lambda x_k.\, e_k$ to that value.

1.3 Injection Functions

Consider $r \in \mathit{Result}$. We know that *Result* is a sum domain, with *Value* as one of its component domains. Further, *Value* is in turn a sum domain, with $\mathbb{Z}$ as one of its component domains. Hence, r may actually be an integer $n \in \mathbb{Z}$. To properly express the integer n as a result value, we must apply two injection functions, from $\mathbb{Z}$ to *Value* , then from *Value* to *Result* :

$$r = \lfloor in_{\mathit{Value} \to \mathit{Result}}(in_{\mathbb{Z} \to \mathit{Value}}(n)) \rfloor$$

However, this is far too wordy, so we will simply use the notation $r = \mathbb{Z}(n) \in \mathit{Result}$ to refer to an integer n in the domain of results. The domain we are referring to and the injection functions that we omit will always be clear from the context.

1.4 Semantics

The semantics of primitive values and variables is:

$$\begin{array}{rcl} \mathcal{C}[\![n]\!]\rho & = & \mathbb{Z}(n) \\ \mathcal{C}[\![\mathsf{true}]\!]\rho & = & \mathbb{T}(\mathit{true}) \\ \mathcal{C}[\![\mathsf{false}]\!]\rho & = & \mathbb{T}(\mathit{false}) \\ \mathcal{C}[\![x]\!]\rho & = & \rho(x) \end{array}$$

The semantics for pairs is:

$$\begin{array}{rcl} \mathcal{C}[\![\mathsf{fst}\ e]\!] & = & \mathit{ecase}\ \ \mathcal{C}[\![e]\!]\rho\ \mathit{of}\ \mathit{Pair}(p) \Rightarrow \pi_1(p) \\ \mathcal{C}[\![\mathsf{snd}\ e]\!] & = & \mathit{ecase}\ \ \mathcal{C}[\![e]\!]\rho\ \mathit{of}\ \mathit{Pair}(p) \Rightarrow \pi_2(p) \\ \mathcal{C}[\![\langle e_0, e_1 \rangle]\!]\rho & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \mathit{Value}(v_1) \Rightarrow \\ & & \quad \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho\ \mathit{of}\ \mathit{Value}(v_2) \Rightarrow (v_1, v_2) \end{array}$$

The semantics for functions is:

$$\begin{array}{rcl} \mathcal{C}[\![e_0 e_1]\!]\rho & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \mathit{Function}(f) \Rightarrow \\ & & \quad \mathit{ecase}\ \ \mathcal{C}[\![e_1]\!]\rho\ \mathit{of}\ \mathit{Value}(v) \Rightarrow f\ v \\ \mathcal{C}[\![\lambda x.\, e]\!]\rho & = & \lambda v \in \mathit{Value}\ .\ \mathcal{C}[\![e]\!]\rho[x \to v] \\ \mathcal{C}[\![\mathsf{rec}\ y\ (\lambda x.\ e)]\!]\rho & = & \mathit{fix}(\lambda f \in \mathit{Function}\ .\ \mathcal{C}[\![\lambda x.e]\!]\rho[y \to f]) \end{array}$$

For the other constructs:

$$\begin{array}{rcl} \mathcal{C}[\![\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]\!]\rho & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \mathit{Value}(v) \Rightarrow \mathcal{C}[\![e_1]\!]\rho[x \to v] \\ \mathcal{C}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho & = & \mathit{ecase}\ \ \mathcal{C}[\![e_0]\!]\rho\ \mathit{of}\ \mathbb{T}(\mathit{true}) \Rightarrow \mathcal{C}[\![e_1]\!]\rho \\ & & \qquad\qquad\qquad\quad \mid \mathbb{T}(\mathit{false}) \Rightarrow \mathcal{C}[\![e_2]\!]\rho \end{array}$$

1.5 Example

We have previously seen that the call-by-value semantics of an application may not terminate if the evaluation of its argument does not terminate. We will try a slightly different example. First consider a recursive function that calls itself (hence its execution doesn't terminate):

$$\mathsf{rec}\ y\ (\lambda x.\ yx)$$

Now consider we pass this recursive procedure as parameter to the constant function zero, which returns 0 regardless of its argument:

$$(\lambda x.\, 0)(\mathsf{rec}\ y\ (\lambda x.\ yx))$$

Note that the evaluation of the argument of this application terminates, but it evaluates to an abstraction which represents a computation that doesn't terminate. Since we the evaluation of the argument terminates, we expect the final answer to be 0 (and not $\perp$). Let's check that using denotational semantics:

$$
\begin{aligned}
\mathcal{C}[\![(\lambda x.\, 0)\,\overbrace{(\mathsf{rec}\ y\ (\lambda x.\ yx))}^{A}]\!]\rho &= \mathit{ecase}\ \ \mathcal{C}[\![\lambda x.\, 0]\!]\rho\ \mathit{of}\ \mathit{Function}(f) \Rightarrow \\
&\qquad \mathit{ecase}\ \ \mathcal{C}[\![A]\!]\rho\ \mathit{of}\ \mathit{Value}(v') \Rightarrow f\ v' \\
&= \mathit{ecase}\ \ \lambda v.\, 0\ \mathit{of}\ \mathit{Function}(f) \Rightarrow \\
&\qquad \mathit{ecase}\ \ \mathcal{C}[\![A]\!]\rho\ \mathit{of}\ \mathit{Value}(v') \Rightarrow f\ v' \\
&= \mathit{ecase}\ \ \mathcal{C}[\![A]\!]\rho\ \mathit{of}\ \mathit{Value}(v') \Rightarrow (\lambda v.\, 0)(v')
\end{aligned}
$$

We know from our definition of A that:

$$\mathcal{C}[\![A]\!]\rho = \mathit{fix}(\lambda f.\, \mathcal{C}[\![\lambda x.\, yx]\!]\rho[y \to f]) = (\lambda v.\, \perp)$$

Therefore:

$$
\begin{aligned}
\mathcal{C}[\![(\lambda x.\, 0)A]\!]\rho &= \mathit{ecase}\ \ (\lambda v.\, \perp)\ \mathit{of}\ \mathit{Value}(v') \Rightarrow (\lambda v.\, 0)(v') \\
&= (\lambda v.\, 0)(\lambda v.\, \perp) = 0
\end{aligned}
$$

Thus the denotational semantics gives us the result we expected.