Andrew Myers

The language F_0 presented in class is a lazy, call-by-name, untyped functional language. However, with minor tweaks to the semantics, it can be converted into an strict, call-by-value language like Scheme, in which function arguments are evaluated eagerly.

1 Syntax

An expression is represented by the keyword *expr*. A program is an expression containing no free variables. Suppose n denotes an integer literal, x denotes a variable name, and e_i denotes an expression. Expressions are then defined by the following grammar:

$$\begin{array}{rcl} \mathit{expr} & ::= & n \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \mid x \mid (\mathit{unop}\ e_0) \mid (\mathit{binop}\ e_0\ e_1) \mid (\mathsf{if}\ e_0\ e_1\ e_2) \\ & \mid & (\mathsf{cons}\ e_0\ e_1) \mid (\mathsf{fn}\ x\ e) \mid (\mathsf{call}\ e_0\ e_1) \mid (\mathsf{rec}\ x\ e) \\ \mathit{binop} & ::= & + \mid * \mid - \\ \mathit{unop} & ::= & \mathsf{hd} \mid \mathsf{tl} \mid \mathsf{zero?} \mid \mathsf{unit?} \end{array}$$

2 Structural Operational Semantics

A single-step operational semantics for F_0 can be constructed without difficulty. First, since an operational semantics rewrites expressions into other expressions, we need to specify what expressions are considered to be normal forms, or values:

$$\mathit{value} ::= n \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \mid (\mathsf{cons}\ e_0\ e_1) \mid (\mathsf{fn}\ x\ e)$$

By examining this definition, we can see that lists in F_0 are evaluated lazily: the head and tail of a cons value are allowed to be arbitrary expressions. The bodies of function values (fn) are also evaluated lazily, which is more typical.

In the following rewrite rules, the letter v is used to denote a value. These rules use substitution to eliminate all free variables, so no rule is given for rewriting a variable name: there is no environment. We begin by presenting a semantics in which computation becomes stuck if there is a type error.

Arithmetic is evaluated in a strict fashion, left to right:

$$\frac{e_0 \rightarrow_1 e_0'}{(\mathit{binop}\ e_0\ e_1) \rightarrow_1 (\mathit{binop}\ e_0'\ e_1)} \qquad \frac{e_1 \rightarrow_1 e_1'}{(\mathit{binop}\ n\ e_1) \rightarrow_1 (\mathit{binop}\ n\ e_1')} \qquad \frac{}{(\mathit{binop}\ n_0\ n_1) \rightarrow_1 v_0\ \mathit{binop}\ n_1}$$

$$\frac{e_0 \rightarrow_1 e_0'}{(\mathit{unop}\ e_0) \rightarrow_1 \mathit{unop}\ e_0'} \qquad \frac{v_0 = 0}{(\mathsf{zero?}\ v_0) \rightarrow_1 \#\mathsf{t}} \qquad \frac{v_0 \neq 0}{(\mathsf{zero?}\ v_0) \rightarrow_1 \#\mathsf{f}}$$

$$\frac{v_0 = \#\mathsf{u}}{(\mathsf{unit?}\ v_0) \rightarrow_1 \#\mathsf{t}} \qquad \frac{v_0 \neq \#\mathsf{u}}{(\mathsf{unit?}\ v_0) \rightarrow_1 \#\mathsf{f}}$$

The if expression evaluates its second two arguments lazily:

$$\frac{e_0 \rightarrow_1 e_1}{(\mathsf{if}\ e_0\ e_1\ e_2) \rightarrow_1 (\mathsf{if}\ e_0'\ e_1\ e_2)} \qquad \frac{}{(\mathsf{if}\ \#\mathsf{t}\ e_1\ e_2) \rightarrow_1 e_1} \qquad \frac{}{(\mathsf{if}\ \#\mathsf{f}\ e_1\ e_2) \rightarrow_1 e_2}$$

Evaluation of list (cons cells) is done only when the head and tail components are extracted.

$$\frac{}{(\mathsf{hd}\ (\mathsf{cons}\ e_0\ e_1)) \rightarrow_1 e_0} \qquad \frac{}{(\mathsf{tl}\ (\mathsf{cons}\ e_0\ e_1)) \rightarrow_1 e_1}$$

A call expression causes a function call. The function argument is evaluated first; the argument expression is substituted into the function body without evaluation. Finally, the rec expression, which is used to construct recursively defined values, is evaluated by unrolling the expression by one level, using substitution.

$$\frac{e_0 \rightarrow_1 e_0'}{(\mathsf{call}\ e_0\ e_1)) \rightarrow_1 (\mathsf{call}\ e_0'\ e_1} \qquad \frac{}{(\mathsf{call}\ (\mathsf{fn}\ x\ e_0)\ e_1) \rightarrow_1 e_0'[e_1/x]} \qquad \frac{}{(\mathsf{rec}\ x\ e) \rightarrow_1 e[(\mathsf{rec}\ x\ e)/x]}$$

We can augment this semantics to halt the program with a result of error when a dynamic type error occurs, rather than simply getting stuck. If error is added as an allowed kind of *value*, then the following rules will generate a result of error when dynamic type errors occur:

$$\frac{v_0 \notin Z}{(\mathit{binop}\ v_0\ e_1) \rightarrow_1 \mathsf{error}} \qquad \frac{v_1 \notin Z}{(\mathit{binop}\ n_0\ v_1) \rightarrow_1 \mathsf{error}}$$

$$\frac{\neg\exists e_0, e_1\ .\ v = (\mathsf{cons}\ e_0\ e_1)}{(\mathsf{hd}\ v) \rightarrow_1 \mathsf{error}} \qquad \frac{\neg\exists e_0, e_1\ .\ v = (\mathsf{cons}\ e_0\ e_1)}{(\mathsf{tl}\ v) \rightarrow_1 \mathsf{error}}$$

$$\frac{v \notin \{\#\mathsf{t}, \#\mathsf{f}\}}{(\mathsf{if}\ v\ e_0\ e_1) \rightarrow_1 \mathsf{error}}$$

More rules are needed to propagate error values through the evaluation:

$$\frac{}{(\mathit{binop}\ \mathsf{error}\ e_1) \rightarrow_1 \mathsf{error}} \qquad \frac{}{(\mathit{binop}\ n\ \mathsf{error}) \rightarrow_1 \mathsf{error}} \qquad \frac{}{(\mathit{unop}\ \mathsf{error}) \rightarrow_1 \mathsf{error}}$$

$$\frac{}{(\mathsf{if}\ \mathsf{error}\ e_0\ e_1) \rightarrow_1 \mathsf{error}} \qquad \frac{}{(\mathsf{call}\ \mathsf{error}\ e_1) \rightarrow_1 \mathsf{error}}$$

3 Denotational Semantics

First, we need to construct the domains that the denotational semantics will map expressions to. We can define these domains using *domain equations* as follows.

$$\begin{array}{rcl}
r \in \mathit{Result} & = & (\mathit{Value} + \mathit{Error})_\perp \\
v \in \mathit{Value} & = & \mathcal{Z} + \mathcal{T} + \mathit{Cons} + \mathit{Function} + \mathit{Unit} \\
f \in \mathit{Function} & = & \mathit{Result} \rightarrow \mathit{Result} \\
n \in \mathcal{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \qquad \text{(a flat CPO)} \\
t \in \mathcal{T} & = & \{\mathit{true}, \mathit{false}\} \qquad \text{(another flat CPO)} \\
c \in \mathit{Cons} & = & \mathit{Result} \times \mathit{Result} \\
\mathit{Error} & = & \mathit{Unit} \\
\rho \in \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Result}
\end{array}$$

Now, how should we translate the domain equations given into definitions for the corresponding domains? We have already seen many examples of similar recursive set definitions, and we can intuitively think of these definitions as being more of the same. For example, we can think of the domain D defined by the equation $D = D \times D + \mathit{Unit}$ as the set defined by the following two inference rules:

$$\frac{d_1 \in D \quad d_2 \in D}{\mathit{in}_1(\langle d_1, d_2\rangle) \in D} \qquad \frac{}{\mathit{in}_2(\mathit{unit}) \in D}$$

This intuition is not precisely correct once we introduce function domains, however. The problem is that we can write function expressions in F_0 that can be described only with an infinite number of applications of the function domain constructor. Our inductive set definition technique only finds the least set satisfying the inference rules; all elements in this set are found by applying the rules a finite number of times. To understand what is going on with function domains, we will need the theory of *information systems*, which will allow us to construct least domains in which the construction rules may be applied an infinite number of times. However, we will not tackle this aspect of the theory yet, and will instead rely on your intuition for what the domain equations mean.

The semantic function $\mathcal{C}$ takes an expression and maps it to a function that produces a result given an environment in which to interpret the free variables of the expression:

$$\mathcal{C}[\![e]\!] : \mathit{Env} \to \mathit{Result}$$

To evaluate a program E, we take $\mathcal{C}[\![E]\!](\lambda x \in \mathit{Var} \,.\, \mathit{error})$; in other words, any free variables in the program will result in an error. The value error is defined as $\lfloor \mathit{in}_{\mathit{Result}\leftarrow\mathit{Error}}(\mathit{in}_{\mathit{Error}\leftarrow\mathit{Unit}}(\mathit{unit}))\rfloor$, where unit is the sole member of the domain Unit, and functions of the form $\mathit{in}_{A\leftarrow B}$ map an element from domain B to the corresponding element of the sum domain A in which B is a component of the sum. For simplicity these functions $\mathit{in}_{A\leftarrow B}$ will not be written down explicitly in the remainder of these semantics, except where there is ambiguity.

We can now define the semantic function $\mathcal{C}$ for each kind of expression. The atomic expressions are straightforward:

$$\begin{array}{rcl}
\mathcal{C}[\![n]\!]\rho & = & n \\
\mathcal{C}[\![\#\mathsf{t}]\!]\rho & = & \mathit{true} \\
\mathcal{C}[\![\#\mathsf{f}]\!]\rho & = & \mathit{false} \\
\mathcal{C}[\![\#\mathsf{u}]\!]\rho & = & \mathit{in}_{\mathit{Result}\leftarrow\mathit{Value}}(\mathit{in}_{\mathit{Value}\leftarrow\mathit{Unit}}(\mathit{unit})) \\
\mathcal{C}[\![x]\!]\rho & = & \rho x
\end{array}$$

Note that the injection functions are applied explicitly for the expression $\#\mathsf{u}$ because the element unit, injected differently, also stands for an error result.

The denotations of the more complex expressions are defined in terms of the denotations of their subexpressions. Here we assume that the case meta-language construct is strict when it is used to construct a function from *Result* to *Result*: the value of $\mathsf{case}\ \bot\ \mathsf{of}\ \ldots\ \mathsf{end}$ is always $\bot$. This constraint preserves monotonicity. A variety of other notational abuses are used below in the various arms of case statements to keep things concise. For example, the notation "$\mathit{Function}(f) \Rightarrow E$" stands for

$$\mathit{in}_{\mathit{Result}\leftarrow\mathit{Value}}(\mathit{in}_{\mathit{Value}\leftarrow\mathit{Function}}(f)) \Rightarrow E$$

$$\begin{array}{rcl}
\mathcal{C}[\![\mathit{binop}\ e_0\ e_1]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e_0]\!]\ \mathsf{of} \\
 & & \quad \mathcal{Z}(n_0) \Rightarrow (\mathsf{case}\ \mathcal{C}[\![e_1]\!]\ \mathsf{of}\ \mathcal{Z}(n_1) \Rightarrow n_0\ \mathit{binop}\ n_1 \mid \mathsf{else}\ \mathit{error}) \\
 & & \quad \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{zero?}\ e]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e]\!]\ \mathsf{of}\ 0 \Rightarrow \mathit{true} \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{unit?}\ e]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e]\!]\ \mathsf{of}\ \mathit{in}_{\mathit{Result}\leftarrow\mathit{Value}}(\mathit{in}_{\mathit{Value}\leftarrow\mathit{Unit}}(u)) \Rightarrow \mathit{true} \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{if}\ e_0\ e_1\ e_2]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e_0]\!]\ \mathsf{of}\ \mathit{true} \Rightarrow \mathcal{C}[\![e_0]\!] \mid \mathit{false} \Rightarrow \mathcal{C}[\![e_1]\!] \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{cons}\ e_0\ e_1]\!]\rho & = & \langle \mathcal{C}[\![e_0]\!]\rho, \mathcal{C}[\![e_1]\!]\rho \rangle \\
\mathcal{C}[\![\mathsf{hd}\ e]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e]\!]\rho\ \mathsf{in}\ \mathit{Cons}(c) \Rightarrow \pi_1(c) \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{tl}\ e]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e]\!]\rho\ \mathsf{in}\ \mathit{Cons}(c) \Rightarrow \pi_2(c) \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho & = & \lambda r \in \mathit{Result}\ .\ \mathcal{C}[\![e]\!](\rho[x \mapsto r]) \\
\mathcal{C}[\![\mathsf{call}\ e_0\ e_1]\!]\rho & = & \mathsf{case}\ \mathcal{C}[\![e_0]\!]\rho\ \mathsf{of}\ \mathit{Function}(f) \Rightarrow f(\mathcal{C}[\![e_1]\!]\rho) \mid \mathsf{else}\ \mathit{error} \\
\mathcal{C}[\![\mathsf{rec}\ x\ e]\!]\rho & = & \mathit{fix}\ \lambda r \in \mathit{Result}\ .\ \mathcal{C}[\![e]\!](\rho[x \mapsto v])
\end{array}$$

It is important to note that writing $\mathcal{C}[\![e]\!]\rho$ in the denotational semantics does not necessarily correspond to complete evaluation of the expression e. For example, a function call may evaluate to an actual value despite the fact that its argument e_1 evaluates to $\bot$. For example, consider the denotation for the program (call (fn x 0) (rec x x)), which we can manipulate as a mathematical term:

$$\begin{aligned}
& \mathcal{C}[\![(\mathsf{call}\ (\mathsf{fn}\ \mathsf{a}\ 0)\ (\mathsf{rec}\ \mathsf{a}\ \mathsf{a}))]\!]\rho \\
= \ & \mathsf{case}\ \mathcal{C}[\![\mathsf{fn}\ \mathsf{a}\ 0]\!]\rho\ \mathsf{of}\ \mathit{Function}(f) \Rightarrow f(\mathcal{C}[\![\mathsf{rec}\ \mathsf{a}\ \mathsf{a}]\!]\rho) \mid \mathsf{else}\ \mathit{error} \\
= \ & \mathsf{case}\ (\lambda r \in \mathit{Result}\ .\ \mathcal{C}[\![0]\!](\rho[\mathsf{a} \mapsto r]))\ \mathsf{of}\ \mathit{Function}(f) \Rightarrow f(\mathcal{C}[\![\mathsf{rec}\ \mathsf{a}\ \mathsf{a}]\!]\rho) \mid \mathsf{else}\ \mathit{error} \\
= \ & (\lambda r \in \mathit{Result}\ .\ 0)(\mathcal{C}[\![\mathsf{rec}\ \mathsf{a}\ \mathsf{a}]\!]\rho) \\
= \ & (\lambda r \in \mathit{Result}\ .\ 0)(\bot_{\mathit{Result}}) \\
= \ & 0
\end{aligned}$$

Thus, the program will terminate with the value 0 even though the sub-expression (rec a a) diverges. Denotational semantics make it very easy to talk about the result of a lazily evaluated expression; in fact, one has to be careful to ensure that expressions are evaluated strictly where this is intended. For example, if we wanted to make lists *strict*, we would define

$$\mathit{Cons} = \mathit{Value} \times \mathit{Value}$$

and change the rule for evaluating cons to eagerly evaluate its elements. Assuming a left-to-right evaluation order, we have this rule:

$$\begin{aligned}
& \mathcal{C}[\![\mathsf{cons}\ e_0\ e_1]\!]\rho = \mathsf{case}\ \mathcal{C}[\![e_0]\!]\rho\ \mathsf{of} \\
& \quad \mathit{in}_{\mathit{Result} \leftarrow \mathit{Value}}(v_0) \Rightarrow (\mathsf{case}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of}\ \mathit{in}_{\mathit{Result} \leftarrow \mathit{Value}}(v_1) \Rightarrow \langle v_0, v_1 \rangle \mid \mathsf{else}\ \mathit{error}) \\
& \quad \mid \mathsf{else}\ \mathit{error}
\end{aligned}$$

When implementing this language, we will want to represent a lazily evaluated expression $\mathcal{C}[\![e]\!]\rho$ by keeping its components $\mathcal{C}[\![e]\!]$ and ρ separate until it is necessary to combine them. This combination of code ($\mathcal{C}[\![e]\!]$) and its environment (ρ) is known as a *closure*.

4 Dynamic vs. Static Scope

This language uses *static scope* based on the block structure of the language, and is said to have *block-structured static scope*. With static scoping, the meaning of an identifier is determined by the point in the program at which the identifier occurs. With *dynamic scoping*, the alternative, the meaning of an identifier is determined by the state of the executing program.

All of the languages we have seen so far in this course (with the exception of the call-by-denotation semantics we saw recently) have had block-structured static scope. Because our denotational semantics makes environments into explicit first-class entities, we can describe alternate scoping mechanisms quite easily.

In most languages, variables are introduced as function arguments and by local variable declarations. The language F_0 does not have local variables, because they can be desugared into a use of fn/call:

$$(\mathsf{let}\ (x\ e_1)\ e_2) \equiv (\mathsf{call}\ (\mathsf{fn}\ x\ e_2)\ e_1)$$

The key rules for determining the scoping system are the rules for evaluating fn and call. In the cases we are interested in, the rule for evaluating let can be determined by composing the rules for call and fn. Ignoring the part of the denotational semantics that catches error conditions, the following definitions from above enforce static scope:

$$
\begin{aligned}
\mathcal{C}[\![\textsf{fn}\ x\ e]\!]\rho &= \lambda r \in \mathit{Result}\ .\ ]\mathcal{C}[\![e]\!](\rho[x \mapsto r]) \\
\mathcal{C}[\![\textsf{call}\ e_0\ e_1]\!]\rho &= \mathcal{C}[\![e_0]\!]\rho(\mathcal{C}[\![e_2]\!]\rho)
\end{aligned}
$$

The key observation here is that any free variables in the function body e obtain their meaning in the environment ρ that exists at the point where the fn expression itself is evaluated.

Dynamic scoping, which was present in most early versions of Lisp and lives on in Perl, takes another approach: an identifier is bound to the the variable existing at the point where the function is *called*, not where the function expression is evaluated. Dynamic scoping is useful because it allows additional arguments to be passed implicitly to a function. However, these hidden arguments are problematic because they violate modularity. The following is an example of a program that evaluates differently in the two apporoaches:

```
(let (delta 2)
  (let (bump (fn x (+ x delta)))
    (let (delta 1) (call bump 2))))
```

With static scope, the identifier delta in the definition of the function bump is bound to the outer variable delta. The program returns the value 4. With dynamic scope, delta is bound to the variable delta that is in scope at the point where the function is called. The program returns the value 3.

To make dynamic scope work, functions need to know the environment at the point where they are applied. Therefore, the domain of functions is changed as follows:

$$\mathit{Function} = \mathit{Result} \to \mathit{Env} \to \mathit{Result}$$

When a function is invoked, a dynamic environment must be supplied so that any free variables in the function body can be interpreted. The rules for evaluating fn and call are as follows:

$$
\begin{aligned}
\mathcal{C}[\![\textsf{fn}\ x\ e]\!]\rho_{\text{lex}} &= \lambda r \in \mathit{Result}, \rho_{\text{dyn}} \in \mathit{Env}\ .\ \mathcal{C}[\![e]\!](\rho_{\text{dyn}}[x \mapsto v]) \\
\mathcal{C}[\![\textsf{call}\ e_0\ e_1]\!]\rho_{\text{dyn}} &= \mathcal{C}[\![e_0]\!]\rho_{\text{dyn}}(\mathcal{C}[\![e_2]\!]\rho_{\text{dyn}}, \rho_{\text{dyn}})
\end{aligned}
$$

One may wonder why early language implementors might have chosen dynamic scope. One reason is the simplicity of implementing functions. The denotational semantics of a function expression state that the lexical environment ρ_{lex} is ignored in the evaluation of a function definition. With dynamic scope, a function (fn x e) is represented by its code $\mathcal{C}[\![e]\!]$, and there is no need to form a closure with the lexical environment.