The language uF presented in class is a call-by-value, left-to-right eager, untyped functional language.

1 Syntax

An expression is represented by the character e. 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:

$$\begin{array}{rcl} b & ::= & n \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \\ e & ::= & b \mid x \mid e_1 \oplus e_2 \mid (\mathsf{if}\ e_0\ e_1\ e_2) \mid \langle e_1, e_2\rangle \mid e_1\ e_2 \mid \\ & \mid & \mathsf{first}\ e \mid \mathsf{rest}\ e \mid \mathsf{fn}\ x\ e \mid (\mathsf{rec}\ x\ e_r) \\ e_r & ::= & \mathsf{fn}\ x\ e \end{array}$$

2 Structural Operational Semantics for uF

The structural operational semantics for uF based on evaluation context can be constructed without difficulty:

$$\begin{array}{rcl} \mathrm{v} & ::= & b \mid \mathsf{fn}\ x\ e \mid \langle v_1, v_2\rangle \\ C & ::= & [\cdot] \mid C \oplus e \mid v \oplus C \mid \mathsf{if}\ C\ e_1\ e_2 \mid \langle C, e\rangle \mid \langle v, C\rangle \\ & \mid & C\ e \mid v\ C \mid \mathsf{first}\ C \mid \mathsf{rest}\ C \end{array}$$

$$\frac{e \to e'}{C[e] \to C[e']} \qquad \frac{}{\mathsf{if}\#\mathsf{t}\ e_1\ e_2 \to e_1}$$

$$\frac{}{\mathsf{if}\#\mathsf{f}\ e_1\ e_2 \to e_2} \qquad \frac{}{(\mathsf{fn}\ x\ e)\ v \to e\{v/x\}}$$

$$\frac{}{\mathsf{first}\ \langle v_1,\ v_2\rangle \to v_1} \qquad \frac{}{\mathsf{rest}\ \langle v_1,\ v_2\rangle \to v_2}$$

$$\frac{}{\mathsf{rec}\ x\ e \to e\{\mathsf{rec}\ x\ e/x\}} \qquad \frac{}{n_1 \oplus n_2 \to n}\ (\text{where } n = n_1 \oplus n_2)$$

Example: the factorial function

$$\begin{array}{l} (\mathsf{rec}\ \mathit{fact}\ (\mathsf{fn}\ x\ \mathsf{if}\ (x < 2)\ 1\ \mathsf{else}\ x * \mathit{fact}(x-1)))\ 3 \\ \to (\mathsf{fn}\ x\ \mathsf{if}\ (x < 2)\ 1\ \mathsf{else}\ x * \mathit{fact}(x-1))\{\mathsf{rec}\ \mathit{fact}\ e/\mathsf{fact}\}\ 3 \\ \to (\mathsf{fn}\ x\ \mathsf{if}\ (x < 2)\ 1\ \mathsf{else}\ x * (\mathsf{rec}\ \mathit{fact}\ldots)(x-1))\ 3 \\ \to (\mathsf{if}\ (x < 2)\ 1\ \mathsf{else}\ 3 * (\mathsf{rec}\ \mathit{fact}\ldots))(3-1) \\ \to (\mathsf{if}\ \#\mathsf{f}\ 1\ \mathsf{else}\ 3 * (\mathsf{rec}\ \mathit{fact}\ldots))(3-1) \\ \to 3 * (\mathsf{rec}\ \mathit{fact}\ldots)(3-1) \\ \to 3 * (\mathsf{rec}\ \mathit{fact}\ldots)(2) \\ \vdots \\ \to 3 * 2 * 1 \end{array}$$

3 Denotational Semantics

In our denotational semantics, we must deal with the possibility of errors. For example, (#u 2) should give an error, as should (x 2), if x is not a unary function. A possible solution would be to make our meaning function partial, but we won't do this; instead, we'll add Error as a possible meaning for programs. So we define the following domains:

$$\begin{aligned}
\mathit{Result} &= (\mathit{Value} + \mathit{Error})_\perp \\
\mathit{Value} &= \mathbb{Z} + \mathbb{T} + \mathbb{U} + \mathit{Value} \times \mathit{Value} + \mathit{Function} \qquad (\mathbb{U} \text{ is Unit.}) \\
\mathit{Function} &= \mathit{Binding} \to \mathit{Result} \\
\mathit{Binding} &= \mathit{Value} \\
\mathbb{Z} &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \qquad \text{(a discrete CPO)} \\
\mathbb{T} &= \{\mathit{true}, \mathit{false}\} \qquad \text{(another discrete CPO)} \\
\mathbb{U} &= \{u\} \\
\mathit{Error} &= \mathbb{U}
\end{aligned}$$

Elements of *Env* will encompass a variable environment and function environment together, so we have

$$\mathcal{C}[\![e]\!] \in \mathit{Env} \to \mathit{Value} + \mathit{Error}$$

The environment will map unbound identifiers to *Error*.

Another new problem to deal with is that these domain definitions are really equations. For example, we have an equation concerning the domain *Value*:

$$\mathit{Value} = \mathbb{Z} + \mathbb{T} + \mathbb{U} + \mathit{Value} \times \mathit{Value} + \mathit{Value} \to (\mathit{Value} + \mathbb{U})$$

In what sense is this a definition of the domain *Value*? We want *Value* to be a fixed point of the domain construction process. We have already seen some examples of similar recursive set definitions, because we have been inductively defining syntactic sets and evaluation relations. 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 $D = D \times D + \mathbb{Z}$, which corresponds to the simple BNF definition $a ::= a \oplus a \mid n$, as the set defined by the following two inference rules:

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

This set D is a fixed point of the rule operator; in fact, as we proved earlier, it is the least fixed point. However, when defining programming language domains, we will not always want to least fixed point. For example, sometimes we might want the *greatest* fixed point, which in this case would include all binary trees, even ones that are allowed to descend infinitely far.

Function domains will also break this intuition. The problem is that we can write function expressions in uF that cannot be described with a finite 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.

For example, if we try to construct a denotational model of the untyped lambda calculus (which uF is a superset of), we encounter the following domain equation: $D = D \to D$, where D is the domain that all lambda calculus expressions evaluate to. This equation lurks at the heart of the uF domain equations above. It would be good enough to find a solution to such an equation to within isomorphism: that is, a domain D such that $D \cong D \to D$, and isomorphisms $\mathit{up} \in (D \to D) \to D$ and $\mathit{down} \in D \to (D \to D)$ that preserve ordering. There is a very simple solution to this equation: $D = \mathbb{U}$, because both $\mathbb{U}$ and $\mathbb{U} \to \mathbb{U}$ have a single element. The corresponding isomorphisms are $\mathit{up} = \lambda x \in \mathbb{U} \to \mathbb{U} \,.\, u$ and $\mathit{down} = \lambda x \in \mathbb{U} \,.\, \lambda y \in \mathbb{U} \,.\, u$. While

this model of the lambda calculus is in a fixed point, it says nothing interesting happens during computation! All functions really are just fancy ways of saying "unit", and unit applied to unit gives unit.

Figuring out how to solve this equation in a way that provides an adequate model for lambda calculus terms is a difficult result for which Dana Scott is justly celebrated. A solution to these equation is known as the *universal domain*, also usually written as $\mathbb{U}$. Unlike the unit domain, it has a structure sufficiently rich that all other domains we are interested in can be embedded within it! To understand what is going on with the universal domain and other recursively defined domains, we will need the theory of *information systems*, which will allow us to construct least domains that are fixed points of the construction rules. We will also need to refine the meaning of $\cdot \to \cdot$ still further. However, we will not tackle this aspect of the theory yet, and will for now simply assert that we can find appropriate solutions to these equations, relying on your intuition for what the domain equations mean. In fact, for the semantics we will define, it usually won't matter *what* the precise solution to the equation is; only that there is one.

The domain *Error* is defined as $\mathbb{U}$, but an error result must be injected up into the *Value* and *Result* domains respectively. The result *error* is defined as $\lfloor in_2(u) \rfloor$. To make this more readable, we will write injection functions, when we write them, in the form $in_{A \leftarrow B}$. Here $A \leftarrow B$ is a long-winded way of writing the index that B has in the sum making up the domain A. Thus, we can write $error = \lfloor in_{Result \leftarrow Error}(u) \rfloor$. For simplicity these functions $in_{A \leftarrow B}$ will not be written down explicitly in the remainder of these semantics, except where there is ambiguity, as in this case where u has two purposes.

Assuming that we have a solution to the domain equations, to obtain the meaning of a program e, we take $\mathcal{C}[\![e]\!](\lambda x \in Var\,.\, \in_2 u)$; in other words, any free variables in the program will result in an error.

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

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

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 sub-expressions:

$$
\begin{array}{rcl}
\mathcal{C}[\![e_1 \oplus e_2]\!]\rho & = & \mathcal{C}[\![e_1]\!]\rho \oplus_{\perp \mathrm{E}} \mathcal{C}[\![e_2]\!]\rho \quad \text{(where } \oplus_{\perp \mathrm{E}} \text{ is the extension of } \oplus \text{ to catch } \perp \text{ and } error\text{)} \\
\mathcal{C}[\![\mathsf{if}\ e_0\ e_1\ e_2]\!]\rho & = & \mathsf{let}\ r = \mathcal{C}[\![e_0]\!]\rho\ . \\
 & & \quad \mathsf{case}\ r\ \mathsf{of} \\
 & & \quad\ v_1\ .\ (\mathsf{case}\ v_1\ \mathsf{of} \\
 & & \quad\quad\ n_1\ .\ error \\
 & & \quad\quad \mid t_2\ .\ \mathsf{if}\ t_2\ \mathsf{then}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{else}\ \mathcal{C}[\![e_2]\!]\rho \\
 & & \quad\quad \mid u_3\ .\ error \\
 & & \quad\quad \mid p_4\ .\ error \\
 & & \quad\quad \mid f_5\ .\ error) \\
 & & \quad \mid e_2\ .\ error
\end{array}
$$

Writing out all these error cases is rather tedious, so we will pretend that we have an improved version

of case in the meta-language:

$$\mathsf{scase}\ e\ \mathsf{of}\ \mathit{pattern}_1\ .\ e_1\ |\ \cdots\ |\ \mathit{pattern}_n\ .\ e_n$$

- strict with respect to e
- any unmatched e's $\Rightarrow$ *error*
- allows patterns—any application of injections and tuple/pair constructions to meta-variables, constants, or other patterns, as in ML pattern-matching.

This new meta-language construct lets us give more compact definitions:

$$\begin{aligned}
\mathcal{C}[\![\mathsf{if}\ e_0\ e_1\ e_2]\!]\rho &= \mathsf{scase}\ \mathcal{C}[\![e_0]\!]\rho\ \mathsf{of} \\
&\qquad \mathit{Value}(\mathcal{T}(t))\ .\ \mathsf{if}\ t\ \mathsf{then}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{else}\ \mathcal{C}[\![e_2]\!]\rho \\
\mathcal{C}[\![\langle e_1,\ e_2\rangle]\!]\rho &= \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of} \\
&\qquad \mathit{Value}(v_1)\ .\ (\mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho\ \mathsf{of}\ \mathit{Value}(v_2)\ .\ \langle v_1,\ v_2\rangle \\
&= \mathsf{scase}\ \langle \mathcal{C}[\![e_1]\!]\rho,\ \mathcal{C}[\![e_2]\!]\rho\rangle\ \mathsf{of}\ \langle \mathit{Value}(v_1),\ \mathit{Value}(v_2)\rangle\ .\ \langle v_1,\ v_2\rangle \\
\mathcal{C}[\![\mathsf{first}\ e]\!]\rho &= \mathsf{scase}\ e\ \mathsf{of}\ \mathit{Value}(\langle v_1,\ v_2\rangle)\ .\ v_1 \\
\mathcal{C}[\![\mathsf{rest}\ e]\!]\rho &= \mathsf{scase}\ e\ \mathsf{of}\ \mathit{Value}(\langle v_1,\ v_2\rangle)\ .\ v_2 \\
\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho &= \lambda v \in \mathit{Value}\ .\ \mathcal{C}[\![e]\!]\rho[x \mapsto v] \\
\mathcal{C}[\![e_1\ e_2]\!]\rho &= \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\ \mathsf{of} \\
&\qquad \mathit{Value}(\mathit{Function}(f))\ .\ \mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho\ \mathsf{of}\ \mathit{Value}(v)\ .\ f(v) \\
\mathcal{C}[\![\mathsf{rec}\ y\ (\mathsf{fn}\ x\ e)]\!]\rho &= \mathit{fix}\ \lambda f \in \mathit{Function}\ .\ \lambda v \in \mathit{Value}\ .\ \mathcal{C}[\![e]\!]\rho[x \mapsto v, y \mapsto f] \\
&\qquad \text{(Function is a pointed cpo, so we can apply fix here.)}
\end{aligned}$$

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 uF does not have local variables, because they can be desugared into an application:

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

The key rules for determining the scoping system are the rules for evaluating fn and function application. 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 (so we are pretending for the moment that $\mathit{Result} = \mathit{Value}$), the following definitions from above enforce static scope:

$$\begin{aligned}
\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho &= \lambda v \in \mathit{Value}\ .\ \mathcal{C}[\![e]\!](\rho[x \mapsto v]) \\
\mathcal{C}[\![e_1\ e_2]\!]\rho &= \mathcal{C}[\![e_1]\!]\rho(\mathcal{C}[\![e_2]\!]\rho)
\end{aligned}$$

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. The following is an example of a program that evaluates differently in the two approaches:

```
let delta = 2 in
  let bump = fn x (x + delta) in
    let delta = 1 in
      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.

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. Given a body of library code, one has no way to know when one is accidentally overriding one of these hidden parameters to the library.

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:

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

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: [1]

$$\begin{aligned}
\mathcal{C}[\![\mathsf{fn}\ x\ e]\!]\rho_{\mathrm{lex}} &= \lambda v \in \mathit{Value}\ .\ \lambda\rho_{\mathrm{dyn}} \in \mathit{Env}\ .\ \mathcal{C}[\![e]\!](\rho_{\mathrm{dyn}}[x \mapsto v]) \\
\mathcal{C}[\![e_1\ e_2]\!]\rho_{\mathrm{dyn}} &= \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho_{\mathrm{dyn}}\ \mathsf{of} \\
&\qquad \mathit{Function}(f).(\mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho_{\mathrm{dyn}}\ \mathsf{of}\ \mathit{Value}(v)\ .\ f v \rho_{\mathrm{dyn}})
\end{aligned}$$

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.

[1] In lecture an scase on a tuple $\langle \mathcal{C}[\![e_1]\!]\rho_{\mathrm{dyn}}, \mathcal{C}[\![e_2]\!]\rho_{\mathrm{dyn}} \rangle$ was used. This approach gives the same meaning as the semantics here for a functional language, but we will need to pin down the evaluation order once we add imperative features and non-local control to the language. So nesting scase expressions is preferred.