CS611 Lecture 12 uF Semantics, Error Checking and Scope 9/26/2001

Scribe: Manpreet Singh (manpreet@cs)
Rohit Ananthakrishna (rohit@cs) Lecturer: Andrew Myers

The language uF presented in class is a call-by-value, left-to-right eager, *untyped* functional language. It is an extension of the *CBV* λ-calculus introduced in the previous lecture.

1. **Syntax of uF:** 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 described by the following BNF grammar

$$b ::= n \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \mid s$$

$$e ::= b \mid x \mid e_1 \oplus e_2 \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \langle e_1, e_2 \rangle \mid \lambda x\, e \mid e_1\ e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \mathsf{rec}\ y(\lambda x\, e)$$

$$v ::= b \mid \lambda x\, e \mid \langle v_1, v_2 \rangle$$

   $\#\mathsf{u}$ denotes a *unit*, $\#\mathsf{t}$ and $\#\mathsf{f}$ denote the booleans *true* and *false* respectively, s represents the strings. $\#\mathsf{u}$ is used when we want to return a dummy value. (This will become clear when we look at error checking in uF.)

2. **SOS to extend CBV semantics with rec:** Note that we have added recursive functions to the *CBV* semantics. So, we extend the *SOS* of *CBV* λ-calculus with *rec*.

$$\frac{}{\mathsf{rec}\ y\ (\lambda x\ e)\ \longmapsto\ \lambda x(e\{\mathsf{rec}\ y\ (\lambda x\ e)/y\})}$$

   We are just unrolling the first recursive call in the above semantics. We illustrate the above semantics with the following **factorial** example

$$\text{let}\ \ \text{fact}\ =\ \mathsf{rec}\ \ f\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else } x * f(x-1) \text{ in fact } 3$$

$$\longmapsto \text{let}\ \ \text{fact}\ =\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else } x*(\ \mathsf{rec}\ f\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else } x*f(x-1))\ (x-1)\ in \text{ fact } 3$$

$$\longmapsto (\lambda x\ (\text{if } x == 0 \text{ then } 1 \text{ else } x * (\mathsf{rec}\ f\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else }\ x * (\mathsf{rec}\ f(x-1))(x-1))\ )\ 3$$

$$\longmapsto_\beta\ (\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\mathsf{rec}\ f\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else }\ x * (\mathsf{rec}\ f(x-1))(3-1))$$

$$\longmapsto\ 3 * (\mathsf{rec}\ f\ \lambda x \text{ if } x == 0 \text{ then } 1 \text{ else }\ x * (\mathsf{rec}\ f(x-1))(2))$$

$$\vdots$$

$$\longmapsto\ 3 * 2 * 1$$

   If we stare at the SOS of **rec** the right hand side of the inference is the same as $(\lambda y \lambda x\ e)(\mathsf{rec}\ y\ (\lambda x\ e))$. That is we have

$$\mathsf{rec}\ y\ (\lambda x\ e)\ \longmapsto\ (\lambda y \lambda x\ e)(\mathsf{rec}\ y\ (\lambda x\ e))$$

   Therefore $(\mathsf{rec}\ y\ (\lambda x\ e))$ is a fixed point of $(\lambda y \lambda x\ e)$. This gives us a clue to how we can translate "rec" into the simple CBV λ calculus. Equivalently,

$$[\![rec\ y\ \lambda x\ e]\!] = FIX_{CBV}(\lambda y \lambda x\ [\![e]\!])$$

3. **Error Checking in uF**

Our previous translation-based semantics for uF didn't deal with stuck configurations in a completely satisfactory way. Once the operations (e.g., left) had been translated into lambda calculus, their behavior in situations that the operational semantics would consider to be stuck is to keep on computing but generating gibberish. For example, if we evaluated the program left 0, we would get the identity function, which makes no sense.

We can perform a translation that models run-time error checking. This means that the translated program must check if operators have been passed with values of the right type. So we tag each of the different possible values permitted in uF and check if an operand has operands having the desired tag values. This tagging corresponds fairly closely to what is done in real untyped languages like Scheme.

Each value is now semantically equivalent to a pair where the first value is the *tag* value and the second operand the actual value. Here is how we translate values:

$$[\![\#\mathsf{u}]\!] = \langle 1, \#\mathsf{u}\rangle$$

$$[\![\#\mathsf{t}]\!] = \langle 2, \#\mathsf{t}\rangle$$

$$[\![\#\mathsf{f}]\!] = \langle 2, \#\mathsf{f}\rangle$$

$$[\![n]\!] = \langle 3, n\rangle$$

$$[\![\langle v_1, v_2\rangle]\!] = \langle 4, \langle [\![v_1]\!], [\![v_2]\!]\rangle\rangle$$

$$[\![\lambda x\ e]\!] = \langle 5, \lambda x\ [\![e]\!]\rangle$$

To the above we add $\langle 0, \#\mathsf{u}\rangle$ which is the tag for *error*. This is returned when a run-time error occurs. The value $\#\mathsf{u}$ in this pair is chosen arbitrarily.

We illustrate the above semantics by looking at two examples.

(i) *Error checking in addition :* The values to "+" must have a tag value of 3 otherwise it is an error.

$$\begin{array}{l}[\![e_1 + e_2]\!] = \text{let } p_1 = [\![e_1]\!] \text{ in}\\ \quad \text{let } p_2 = [\![e_2]\!] \text{ in}\\ \quad\quad \text{let } t_1 = \text{left } p_1 \text{ in}\\ \quad\quad\quad \text{let } t_2 = \text{left } p_2 \text{ in}\\ \quad\quad\quad\quad \text{if } t_1 = 3 \ \wedge\ t_2 = 3 \text{ then } \langle 3, \text{right } p_1 + \text{right } p_2\rangle \text{ else } \langle 0, \#\mathsf{u}\rangle\end{array}$$

(ii) *Error checking in function application :* If we have a function application $(e_1\ e_2)$, we have to ensure that e_1 is indeed a function.

$$\begin{array}{l}[\![e_1\ e_2]\!] \ = \ \text{let } p_1 = [\![e_1]\!] \text{ inlet } p_2 = [\![e_2]\!] \text{ in}\\ \quad \text{let } t_1 = \text{left } p_1 \text{ in}\\ \quad\quad \text{if } t_1 = 5 \text{ then } (\text{right } p_1)\, p_2 \ \text{ else } \langle 0, \#\mathsf{u}\rangle\end{array}$$

What is our notion of correctness for this translation? In addition to the earlier notions of correctness, we add the following: if an evaluation would have gotten stuck in the operational semantics, the translated program should evaluate to $\langle 0, \#\mathsf{u}\rangle$. If S is a representative stuck expression,

$$e \longmapsto^* S \Rightarrow [\![e]\!] \longmapsto^* \langle 0, \#\mathsf{u}\rangle$$

where

$$
\begin{array}{rcl}
S & ::= & \#\mathsf{u}\ v \mid \#\mathsf{t}\ v \mid \#\mathsf{f}\ v \mid n\ v \mid \langle v_1, v_2 \rangle\ v \\
& & \mid \mathsf{if}\ \#\mathsf{u}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \ldots \\
& & \mid \#\mathsf{u} \oplus \#\mathsf{u} \mid \ldots \\
& & \mid \mathsf{left}\ \#\mathsf{u} \mid \mathsf{right}\ \#\mathsf{u} \mid \ldots
\end{array}
$$

4. **Scope Rules:** We have been looking at languages with *block-structured static scope*, in which identifiers are bound to variables based on the static (lexical) structure of the program. With static scope, the binding of a variable is determined by the point in the program at which the identifier *occurs*. The *IMP* language that we saw in this course has block-structured static scope. Because our translational semantics makes environments into explicit first-class entities, we can describe alternate scoping mechanisms quite easily. With *dynamic scope*, the alternative, an identifier is bound to the variable existing at the point where the function is *called*, not where the function expression is evaluated. Thus, the meaning of an identifier is determined by the state of the executing program.

   The following example evaluates differently in the two approaches:

$$
\begin{array}{l}
\text{let delta} = 2 \text{ in} \\
\quad \text{let bump} = \lambda x\ (x + \text{delta}) \text{ in} \\
\quad\quad \text{let delta} = 1 \text{ in} \\
\quad\quad\quad \text{bump } 2
\end{array}
$$

   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 the scope at the point where the function is called. The program returns the value 3.

5. **Semantics to capture the scope rules:** The store that we saw in IMP maps locations to their values. An environment is a map from strings to other things. We will explicitly represent the environments as uF term ρ that maps variable names (as strings) to their values. That is $\rho(\text{“}x\text{”})$ returns the value of x in ρ.

   1. *Static scope:*
      The denotational semantics for static scope are

$$\mathcal{D}[\![x]\!] = \lambda\rho\ \rho(\text{“}x\text{”})$$

$$\text{Applying both sides to an arbitrary } \rho, \text{ we get} \quad \mathcal{D}[\![x]\!]\rho = \rho(\text{“}x\text{”})$$

$$\mathcal{D}[\![b]\!] = \lambda\rho\ b$$

$$\mathcal{D}[\![\text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!] = \lambda\rho \text{ if } \mathcal{D}[\![e_0]\!]\rho \text{ then } \mathcal{D}[\![e_1]\!]\rho \text{ else } \mathcal{D}[\![e_2]\!]\rho$$

      Semantics of function declaration and *let* are more complicated. In order to define this we use a new function extend.

$$\mathit{extend} \ \equiv\ \lambda\rho\lambda x\lambda v(\lambda z \text{ if } x = z \text{ then } v \text{ else } \rho(z))$$

      Intuitively, *extend* takes in an environment ρ, variable x and value v and assigns v to x and returns a new environment in which this change has been reflected.
      Using this,

$$\mathcal{D}[\![\lambda x\ e]\!]\rho = \lambda y\ \mathcal{D}[\![e]\!](\text{extend } \rho\ \text{“}x\text{”}\ y)$$

$$\mathcal{D}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho = (\text{let } y = \mathcal{D}[\![e_1]\!]\rho \text{ in } (\mathcal{D}[\![e_2]\!](\text{extend } \rho\ \text{“}x\text{”}\ y)))$$

$$\mathcal{D}[\![e_1\ e_2]\!]\rho = (\mathcal{D}[\![e_1]\!]\rho)(\mathcal{D}[\![e_2]\!]\rho)$$

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 function expression itself is evaluated.
Now, we apply these rules to an example and show how it evaluates under static scope.

$$\begin{aligned}
e &\equiv \text{(let delta = 2 in let bump = } \lambda x\ (x\ +\ \text{delta) in let delta = 1 in bump 2)} \\
e_1 &\equiv \text{(let bump = } \lambda x\ (x\ +\ \text{delta) in let delta = 1 in bump 2)} \\
e_2 &\equiv \text{(let delta = 1 in bump 2)} \\
e_3 &\equiv \text{(bump 2)} \\
e_4 &\equiv (\lambda x\ (x\ +\ \text{delta})) \\
e_5 &\equiv \lambda w\ (\mathcal{D}\ [\![z + delta]\!]\ \text{(extend } \rho_1\ \text{“z” w))} \qquad (1) \\
\rho_1 &\equiv \text{extend } \rho\ \text{“delta” 2} \\
\rho_2 &\equiv \text{extend } \rho_1\ \text{“bump”} \ e_5 \\
\rho_3 &\equiv \text{extend } \rho_2\ \text{“delta” 1}
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}\ [\![e]\!]\ \rho &\longmapsto \text{(let y = } (\mathcal{D}[\![2]\!]\rho) \text{ in } (\mathcal{D}\ [\![e_1]\!]\ \text{(extend } \rho\ \text{“delta” y)))} \\
&\longmapsto \text{(let y = 2 in } (\mathcal{D}\ [\![e_1]\!]\ \text{(extend } \rho\ \text{“delta” y)))} \\
&\longmapsto (\mathcal{D}\ [\![e_1]\!]\ \rho_1) \\
&\longmapsto \text{(let y = } (\mathcal{D}[\![e_4]\!]\rho_1) \text{ in } (\mathcal{D}\ [\![e_2]\!]\ \text{(extend } \rho_1\ \text{“bump” y))} \\
&\longmapsto \text{(let y = } e_5 \text{ in } (\mathcal{D}\ [\![e_2]\!]\ \text{(extend } \rho_1\ \text{“bump” y))} \\
&\longmapsto (\mathcal{D}\ [\![e_2]\!]\ \rho_2) \\
&\longmapsto \text{(let y = } (\mathcal{D}[\![1]\!]\rho_2) \text{ in } (\mathcal{D}\ [\![e_3]\!]\ \text{(extend } \rho_2\ \text{“delta” y)))} \\
&\longmapsto \text{(let y = 1) in } (\mathcal{D}\ [\![e_3]\!]\ \text{(extend } \rho_2\ \text{“delta” y)))} \\
&\longmapsto (\mathcal{D}\ [\![e_3]\!]\ \rho_3) \\
&\longmapsto (\mathcal{D}\ [\![bump]\!]\ \rho_3)\ (\mathcal{D}\ [\![2]\!]\ \rho_3) \\
&\longmapsto (\mathcal{D}\ [\![bump]\!]\ \rho_3)\ 2 \\
&\longmapsto \rho_3(\text{“bump”})\ 2 \\
&\longmapsto e_5\ 2 \\
&\longmapsto (\lambda w\ (\mathcal{D}\ [\![z + delta]\!]\ \text{(extend } \rho_1\ \text{“z” w))}\ 2) \\
&\longmapsto (\mathcal{D}\ [\![z + delta]\!]\ \text{(extend } \rho_1\ \text{“z” 2))} \\
&\longmapsto 4
\end{aligned}$$

2. *Dynamic Scope:* To make dynamic scope work, functions need to know the environment at the point where they are applied. When a function is invoked, a dynamic environment ρ_{dyn} must be supplied so that any free variables in the function body can be interpreted. This is captured by the following translation:

$$\mathcal{D}[\![\lambda x\ e]\!]\rho_{lex} = \lambda x \lambda \rho_{dyn}\ \mathcal{D}[\![e]\!]\rho_{dyn}$$

So, we discard the environment of static scope ρ_{lex}, take in the current run-time environment ρ_{dyn} and then use the same emantics as static scope, under the new environment ρ_{dyn}.
The semantics for function application is as follows:

$$\mathcal{D}[\![e_1\ e_2]\!]\rho = (\mathcal{D}[\![e_1]\!]\rho)\ (\mathcal{D}[\![e_2]\!]\rho)\ \rho$$

The idea behind this is that we first find $\mathcal{D}[\![e_1]\!]$ under the current environment ρ to obtain, say, e_1'. Next we find $\mathcal{D}[\![e_2]\!]$ under ρ to get e_2'. Note that we expect e_1' to be a function that takes

a binding and a dynamic environment ρ_{dyn} as input and returns the result. Since $(e_1\ e_2)$ is a function application, we must also pass the current environment, which would take the place of ρ_{dyn} in the body of e_1'. Also, we expect e_2' to be a binding. Hence, we apply e_1' to e_2' and ρ.
Now we see how the expression e defined on previous page translates in dynamic scope.

$$
\begin{aligned}
e &\equiv \text{(let delta = 2 in let bump = } \lambda x\ (x\ +\ \text{delta) in let delta = 1 in bump 2)} \\
e_1 &\equiv \text{(let bump = } \lambda x\ (x\ +\ \text{delta) in let delta = 1 in bump 2)} \\
e_2 &\equiv \text{(let delta = 1 in bump 2)} \\
e_3 &\equiv \text{(bump 2)} \\
e_4 &\equiv (\lambda x\ (x\ +\ \text{delta})) \\
e_6 &\equiv \lambda w\ \lambda \rho_{dyn}\ (\mathcal{D}\ [\![w + delta]\!]\ \rho_{dyn}) \\
\rho_1 &\equiv \text{extend } \rho \text{ “delta” } 2 \\
\rho_4 &\equiv \text{extend } \rho_1 \text{ “bump” } e_6 \\
\rho_5 &\equiv \text{extend } \rho_4 \text{ “delta” } 1
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D}[\![e]\!]\ \rho &\longmapsto \text{(let y = } (\mathcal{D}[\![2]\!]\rho) \text{ in } (\mathcal{D}\ [\![e_1]\!]\ \text{(extend } \rho \text{ “delta” y)))} \\
&\longmapsto \text{(let y = 2 in } (\mathcal{D}\ [\![e_1]\!]\ \text{(extend } \rho \text{ “delta” y)))} \\
&\longmapsto (\mathcal{D}\ [\![e_1]\!]\ \rho_1) \\
&\longmapsto \text{(let y = } (\mathcal{D}[\![e_4]\!]\rho_1) \text{ in } (\mathcal{D}\ [\![e_2]\!]\ \text{(extend } \rho_1 \text{ “bump” y))} \\
&\longmapsto \text{(let y = } e_6 \text{ in } (\mathcal{D}\ [\![e_2]\!]\ \text{(extend } \rho_1 \text{ “bump” y))} \\
&\longmapsto (\mathcal{D}\ [\![e_2]\!]\ \rho_4) \\
&\longmapsto \text{(let y = } (\mathcal{D}[\![1]\!]\rho_4) \text{ in } (\mathcal{D}\ [\![e_3]\!]\ \text{(extend } \rho_4 \text{ “delta” y)))} \\
&\longmapsto \text{(let y = 1) in } (\mathcal{D}\ [\![e_3]\!]\ \text{(extend } \rho_4 \text{ “delta” y)))} \\
&\longmapsto (\mathcal{D}\ [\![e_3]\!]\ \rho_5) \\
&\longmapsto (\mathcal{D}\ [\![bump]\!]\ \rho_5)\ (\mathcal{D}\ [\![2]\!]\ \rho_5)\ \rho_5 \\
&\longmapsto (\mathcal{D}\ [\![bump]\!]\ \rho_5)\ \ 2\ \ \rho_5 \\
&\longmapsto \rho_5(\text{“bump”})\ 2\ \rho_5 \\
&\longmapsto e_6\ 2\ \rho_5 \\
&\longmapsto \mathcal{D}\ [\![2 + delta]\!]\ \rho_5 \\
&\longmapsto 3
\end{aligned}
$$

6. **Dynamic Scope vs Static Scope:**
Dynamic scoping was present in most early versions of List and lives on in Perl and Python. 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. Thus, with dynamic scope, the representation of a function value does not need to contain any information about environments. Dynamic scoping is also useful because it allows *additional arguments* to be passed implicitly to a function by changing its free variables. This makes it more versatile than static scope. 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. Also, static scoping runs *faster* than dynamic scoping. This is so because in static scoping, the memory location at which each variable is stored can have a fixed offset. On the other hand, in dynamic scope, the representation of a function value does not contain any information about environments.