CS611 Lecture 27 **Dynamic Allocation** 5 November 2003

Scribe: Mingsheng Hong Lecturer: Radu Rugina

In this lecture, we extend FUN by adding dynamic allocation. We will make the semantics of this feature similar to that of ML. We call this new language FUN!.

1 Language Syntax and Informal Semantics

We extend the syntax of FUN as follows:

$$e ::= \ldots \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid e_1; e_2 \mid \mathsf{unit}$$

Informally, these new constructs have the following meaning:

$\mathsf{ref}\ e$	Dynamically create a location and initialize it with the value of e;
$!e$	Dereference location e and return the value of the location to which it corresponds;
$e_1 := e2$	Update the location e_1 with the value of e_2 and return unit, whose meaning will be explained below;
$e_1; e_2$	Evaluate e_1 and then e_2, the value returned by the whole expression will be the value returned by evaluating the subexpression e_2;
unit	This construct is used only as the value returned by the assignment construct, denoting the fact that the assignment expression is evaluated only for its side-effect.

Now let us try to write the factorial function using these new constructs.

$$\begin{array}{l} \mathsf{let}\ \mathit{fact} = \lambda x.\ \mathbf{if}\ (!x = 0)\ \mathbf{then}\ 1\ \mathbf{else}\ (!x) * \mathit{fact}(\mathsf{ref}\ \ (!x - 1))\ \mathsf{in} \\ \mathsf{let}\ x =\ \ \mathsf{ref}\ 0\ \ \mathsf{in}\ (x := 3; \mathit{fact}(x)) \end{array}$$

Here x is a reference to some heap location that is allocated by the ref construct.

2 Operational Semantics

Here we present the small-step operational semantics for language FUN!. In evaluating expressions, we will take into account the state, which is a mapping from heap locations to values. Therefore, the evaluation will take the following form:

$$\langle e, \sigma \rangle \rightarrow_1 \langle e', \sigma' \rangle$$

Note that the concept of state here is different from that in IMP. Essentially, a state in IMP is statically bound, which means it contains only the locations corresponding to all the variables that show up in the program. So it must be finite. However, here in FUN!, a state is statically unbound, because every time a new location is allocated by ref, the state will expand its set of locations to include the new location also. Therefore, the number of locations a state contains may potentially go to infinity.

The evaluation of all the constructs common to FUN and FUN! is straightforward. The rules of FUN! are similar to those of FUN, except that they must keep track of the state changes to take into account of the side-effects. For example, we will compare one rule of the evaluation of a pair construct in FUN and FUN! as follows:

$$\frac{e_0 \rightarrow_1 e_0'}{\langle e_0, e_1 \rangle \rightarrow_1 \langle e_0', e_1 \rangle}\ (\text{FUN}) \qquad \frac{\langle e_0, \sigma \rangle \rightarrow_1 \langle e_0', \sigma' \rangle}{\langle \langle e_0, e_1 \rangle, \sigma \rangle \rightarrow_1 \langle \langle e_0', e_1 \rangle, \sigma' \rangle}\ (\text{FUN!})$$

So now, we focus on the semantic rules for those new constructs given at the beginning of this lecture. For each of them, there will be several rules. First, there will be rules which evaluate the subexpressions to values. Then, there will be rules which use these values and perform the appropriate state changes and return the appropriate values. For instance, the evaluation of $\mathsf{ref}\ e$ first evaluates the subexpresson e to a value v, then evaluates $\mathsf{ref}\ v$. The rules in the first category are similar to their counterparts in FUN and we will omit them; we will only show the rules in the second category.

We redefine values as:

$$v ::= n \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\ e \mid \langle v,\, v\rangle \mid l \mid \mathsf{unit}$$

- $\mathsf{ref}\ e$

$$\frac{}{\langle \mathsf{ref}\ v,\, \sigma\rangle \to_1 \langle l,\, \sigma \cup \{l \mapsto v\}\rangle}\ l \notin dom(\sigma)$$

  We create a new location l, and make sure that l does not belong to the existing set of locations in state σ. Then we initialize this location with value v, and expand the state to include this new location.

- $!e$

$$\frac{}{\langle !l,\, \sigma\rangle \to_1 \langle \sigma(l),\, \sigma\rangle}\ l \in dom(\sigma)$$

  We look up the location l in state σ, and return the value corresponding to this location. The state does not change after this evaluation.

- $e_1 := e_2$

$$\frac{}{\langle l := v,\, \sigma\rangle \to_1 \langle \mathsf{unit},\, \sigma[l \mapsto v]\rangle}\ l \in dom(\sigma)$$

  This expression is evaluated only for its side effect. That is, we change the state such that it binds the new value v to the location l. The return value is the special unit value.

- $e_1; e_2$

$$\frac{}{\langle v; v',\, \sigma\rangle \to_1 \langle v',\, \sigma\rangle}$$

  The expression e_1 is evaluated only for the side effect. After we evaluate the two subexpressions e_1 and e_2 to some values v and v' respectively, we return v' as the result.

3 Denotational Semantics

Since now we add in states, heap locations and the unit value, we will extend the semantic domains for FUN as follows.

$$\begin{aligned}
\mathbb{U} &= \{\mathsf{unit}\}\\
\mathit{Value} &= \mathbb{Z} + \mathbb{T} + \mathbb{U} + \mathit{Pair} + \mathit{Function} + \mathit{Location}\\
\mathit{Pair} &= \mathit{Value} \times \mathit{Value}\\
\mathit{Function} &= \mathit{Value} \to \mathit{State} \to \mathit{Result}\\
\mathit{Result} &= (\mathit{Value} \times \mathit{State} + \mathit{Error})_\perp\\
\mathit{State} &= \mathit{Location} \rightharpoonup \mathit{Value}
\end{aligned}$$

In FUN, $\mathcal{C}[\![e]\!] \in \mathit{Env} \to \mathit{Result}$, where $\mathit{Env} : \mathit{Var} \to \mathit{Value}$. Now we keep the same definition for Env, and change the denotation to be:

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

This is to say, the denotation $\mathcal{C}[\![e]\!]$ takes an environment (containing the current values of all stack variables) and a state (containing the current values of all heap locations/variables).

After we define these domains, we can now define the denotational semantics for all the types of expressions. Similar to what we did in FUN, we will use the *ecase* construct to handle both errors and non-termination.

$$
\begin{array}{rcl}
\mathcal{C}[\![n]\!]\rho\sigma & = & (n, \sigma) \\[1em]
\mathcal{C}[\![x]\!]\rho\sigma & = & (\rho(x), \sigma) \\[1em]
\mathcal{C}[\![\textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2]\!]\rho\sigma & = & \mathit{ecase}\ \mathcal{C}[\![e_0]\!]\rho\sigma\ \mathit{of}\ (\mathbb{T}(\textsf{true}), \sigma') \Rightarrow \mathcal{C}[\![e_1]\!]\rho\sigma' \\
 & & \qquad\qquad\qquad\quad \mid (\mathbb{T}(\textsf{false}), \sigma') \Rightarrow \mathcal{C}[\![e_2]\!]\rho\sigma' \\[1em]
\mathcal{C}[\![e_0\ e_1]\!]\rho\sigma & = & \mathit{ecase}\ \mathcal{C}[\![e_0]\!]\rho\sigma\ \mathit{of}\ (\mathit{Function}(f), \sigma') \Rightarrow \\
 & & \mathit{ecase}\ \mathcal{C}[\![e_1]\!]\rho\sigma'\ \mathit{of}\ (\mathit{Value}(v), \sigma'') \Rightarrow f\ v\ \sigma'' \\[1em]
\mathcal{C}[\![\lambda x.\ e]\!]\rho\sigma & = & \lambda v \in \mathit{Value}.\ \lambda \sigma_{dyn} \in \mathit{State}.\ \mathcal{C}[\![e]\!]\rho[x \mapsto v]\sigma_{dyn} \\[1em]
\mathcal{C}[\![\textsf{rec}\ y\ (\lambda x.\, e)]\!]\rho\sigma & = & \mathit{fix}(\lambda f \in \mathit{Function}.\ \mathcal{C}[\![\lambda x.\ e]\!]\rho[y \mapsto f]\sigma)
\end{array}
$$