Scribe: André Allavena, Vadim Grinshpun Lecturer: Andrew Myers

1 Extending uF to uF!

Let's define the language uF! to be a superset of uF that also deals with state (the bang ("!") is used in uF! for dereferencing, hence the name). The grammar is extended as follows:

$$e ::= \{\text{everything from uF}\} \quad | \quad \mathsf{ref} \quad e \quad | \quad !e \quad | \quad e_1 := e_2 \quad | \quad l$$

The l is a location that can be assigned a value. In the assignment $e_1 := e_2$, e_1 must evaluate to a location and e_2 to a value. A program cannot contain any location expressions, although they may be generated during execution of the operational semantics.

We also need to define the following domains to deal with the introduction of state:

$$\mathbf{Storable} = \mathbf{Values} + \mathbf{Error}$$
$$\mathbf{Store} = \mathbf{Location} \longrightarrow \mathbf{Storable}$$

Above, **Location** is a c.p.o. and **Error** represents an unmapped location.

2 A lazy evaluation scheme

$$\begin{array}{l}
\mathsf{let}\ lazyval = \langle \mathsf{ref}\ (\#f), \mathsf{ref}\ (\mathsf{fn}\ u\ e)\rangle\ \mathsf{in} \\
\quad \mathsf{let}\ getval = \mathsf{fn}\ lv\ (\ \mathsf{if}\ (!(\mathsf{first}\ lv)) \quad (!(\mathsf{rest}\ lv)) \quad . \\
\qquad\qquad (\mathsf{let}\ v_1 = (!(\mathsf{rest}\ lv))\#u\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ u_1 = ((\mathsf{first}\ lv)) := \#t)\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{let}\ u_2 = ((\mathsf{rest}\ lv)) := v_1) \\
\qquad\qquad\qquad\quad \mathsf{in}\ v_2 \\
\qquad\qquad \mathsf{in} \\
\qquad\qquad \ldots \\
\qquad\qquad\quad getval(lazyval)
\end{array}$$

The expression e is only evaluated once, no matter how many times $getval(longval)$ is evaluated.

3 Small Step Operational Semantics of uF!

Most expressions in uF! are not aware of state, and therefore are no different from their uF counterparts. Thus, for these expressions we can borrow the semantics from uF, taking care to include the state σ in the configuration, and make sure it is preserved. Therefore, a rule of the form $e \longrightarrow e'$ becomes $(e, \sigma) \longrightarrow (e', \sigma)$. For example,

$$(\text{if } \#t\ e_1\ e_2) \longrightarrow e_1$$

changes to

$$(\text{if } \#t\ e_1\ e_2, \sigma) \longrightarrow (e_1, \sigma)$$

We now extend the evaluation context as follows:

$$C ::= \ldots | \quad \mathsf{ref}\ C \quad | \quad !C \quad | \quad C := e \quad | \quad v ::= C$$

$$v ::= \ldots | \quad l$$

The general rule is

$$\frac{(e, \sigma) \longrightarrow (e', \sigma')}{(C[e], \sigma) \longrightarrow (C[e'], \sigma')}$$

Here are the rest of the rules:

$$(\mathsf{ref}\ v, \sigma) \longrightarrow (l, \sigma[l \mapsto v]) \qquad \text{where } \sigma l = error.$$
$$(!l, \sigma) \longrightarrow (\sigma(l), \sigma)$$
$$(l := v, \sigma) \longrightarrow (\#u, \sigma[l \mapsto v])$$

We do not want to allocate the same location twice. Nothing prevents a location from being stored in a location. To model a language like C where we can explicitly deallocate locations, we can also define an expression $free\ e$ that works as follows:

$$\frac{\sigma l \neq error}{\langle free\ l, \sigma\rangle \ \rightarrow\ \langle \#u, \sigma\rangle}$$

We have chosen to represent stores as fuctions; another way to represents the stores would be to have *Storable* be a list of pairs expression value :

$$\mathbf{Storable} = \langle\langle e_1, v_1\rangle, \langle e_2, v_2\rangle, \ldots\rangle$$

4 CBV Denotational Semantics of uF!

Suppose we are given the functions ρ (the environment) and σ (the store). We can then define the formal denotational semantics of uF!:

$$\begin{array}{l}
\mathcal{C}[\![e]\!] \in \mathbf{Env} \to \mathbf{Store} \to \mathbf{Result} \\
\sigma \in \mathbf{Store} = (\mathbf{Location} \to \mathbf{Value} + \mathbf{Error}) \\
\rho \in \mathbf{Env} = (\mathbf{Var} \to \mathbf{Value} + \mathbf{Error}) \\
f \in \mathbf{Function} = (\mathbf{Value} \to \mathbf{Store} \to \mathbf{Result}) \\
\mathbf{Result} = ((\mathbf{Value} + \mathbf{Error}) \times \mathbf{Store})_\perp \\
\mathbf{Value} = \mathbb{U} + \mathbb{Z} + \mathbb{T} + \mathbf{Pair} + \mathbf{Function} + \mathbf{Location}
\end{array}$$

We should change our "scase" expression to reflect the fact that our generic error is no longer represented by *error* but by the pair $\langle error, \sigma\rangle$. Assuming we have done so, we can define the meaning $\mathcal{C}[\![e]\!]\rho\sigma$ of an expression e inductively as follows:

$$\begin{array}{rcl}
\mathcal{C}[\![n]\!]\rho\sigma & = & \langle n, \sigma\rangle \\
\mathcal{C}[\![e_1 \oplus e_2]\!]\rho\sigma & = & \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(v_1), \sigma'\rangle\ .\ \mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho\sigma'\ \mathsf{of}\ \langle \mathsf{Value}(v_2), \sigma''\rangle. \\
 & & \quad \langle v_1 \oplus_E v_2, \sigma''\rangle \\
\mathcal{C}[\![\mathbf{if}\ e_0\ e_1\ e_2]\!]\rho\sigma & = & \mathsf{scase}\ \mathcal{C}[\![e_0]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(\mathbb{T}(l)), \sigma'\rangle\ . \\
 & & \mathbf{if}\ t\ \mathbf{then}\ \mathcal{C}[\![e_1]\!]\rho\sigma'\ \mathbf{else}\ \mathcal{C}[\![e_2]\!]\rho\sigma' \\
\mathcal{C}[\![\mathsf{fn}\ \ x\ e]\!]\rho\sigma_{lex} & = & \lambda v\sigma_{dyn}\ .\ \mathcal{C}[\![e_0]\!]\rho[x \mapsto v]\sigma_{dyn} \\
\mathcal{C}[\![e_1\ e_2]\!]\rho\sigma & = & \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(\mathsf{Function}(f)), \sigma'\rangle\ .\ \mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho\sigma'\ \mathsf{of} \\
 & & \quad \langle \mathsf{Value}(v), \sigma''\rangle\ .\ fv\sigma'' \\
\mathcal{C}[\![\mathsf{rec}\ y\ (\mathsf{fn}\ x\ e)]\!]\rho\sigma & = & \mathsf{fix}\ \lambda f \in \mathbf{Function}.\lambda v\sigma'\ \mathcal{C}[\![e]\!]\rho[x \mapsto v,\ y \mapsto f]\sigma' \\
\mathcal{C}[\![ref\ e]\!] & = & \mathsf{scase}\ \mathcal{C}[\![e]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(v), \sigma'\rangle\ .\ \mathsf{let}\ l = \mathsf{malloc}(\sigma')\ \mathsf{in}\ \langle l, \sigma'[l \mapsto v]\rangle \\
\mathcal{C}[\![!e]\!] & = & \mathsf{scase}\ \mathcal{C}[\![e]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(\mathsf{Location}(l)), \sigma'\rangle\ .\ \langle \sigma'(l), \sigma'\rangle \\
\mathcal{C}[\![e_1 := e_2]\!] & = & \mathsf{scase}\ \mathcal{C}[\![e_1]\!]\rho\sigma\ \mathsf{of}\ \langle \mathsf{Value}(\mathsf{Location}(l)), \sigma'\rangle\ .\ \ \mathsf{scase}\ \mathcal{C}[\![e_2]\!]\rho\sigma'\ \mathsf{of}\ \langle \mathsf{Value}(v), \sigma''\rangle\ . \\
 & & \quad \langle u, \sigma''[l \mapsto v]\rangle
\end{array}$$

We assume we have a deterministic malloc function which, given a state σ, creates a new location in it and maps it to the value *Error* in σ. We don't care which such function it is.

$$\begin{array}{rcl}
\mathsf{malloc} & : & \mathbf{Store} \ \rightarrow\ \mathbf{Location} \\
\text{s.t.} & & \sigma(\mathsf{malloc}(\sigma)) = \mathsf{error}
\end{array}$$

5 Assignments in IMP

The language IMP allows assignments to variables, not to locations. We can translate an imperative language with mutable variables by a translation like the following.

$$\begin{array}{rcl} x & :: & e \\ \mathcal{D}[\![x]\!] & = & !x \\ \mathcal{D}[\![x := e]\!] & = & x := \mathcal{D}[\![e]\!] \\ \mathcal{D}[\![\mathsf{let}\ x = e_1 \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ x = \mathsf{ref}\ \mathcal{D}[\![e_1]\!] \mathsf{in}\ \mathcal{D}[\![e_2]\!] \end{array}$$