Scribe: Sabina Petride Lecturer: Andrew Myers

1 uF!

We extend **uF** with ML-style reference cells:

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

where $\mathsf{ref}\ e$ creates a new location containing e, $!e$ derefences expression e, and $e_1 := e_2$ updates location e_1 with the value of e_2. Notice that $e_1 := e_2$ is special, since it has side-effects (also called *mutations*). So $e_1; e_2$ evaluates e_1 with or without side-effects, and then evaluates e_2.

For example, consider the following expression:
$\mathsf{let}\ x = \mathsf{ref}\ 1\ \mathsf{in}$
$(x := 2;\ !x)$
It creates a new location containing 1, then the location stores the value 2, and dereferencing returns 2.

Using reference cells, we can model more complicated mutable structures, for example mutable arrays: $\mathsf{let}\ x = \langle \mathsf{ref}\ 1, \langle \mathsf{ref}\ 2, \mathsf{ref}\ 0\rangle\rangle\ \mathsf{in}\ \ldots.$

2 SOS for uF!

Because of possible side-effects, expressions alone are not adequate configurations any more: we need a pair expression-store: e, σ. If Loc is a countable set of locations, then a store σ is a partial function that maps locations to values. An evaluation relation is written as $e, \sigma \mapsto e', \sigma'$, and a final configuration has the form v, σ.

Since expressions in **uF** have no side-effects, the following inference rule shows how to lift **uF** evaluation relation to the **uF!** evaluation relation:

$$\frac{e \overset{uF}{\mapsto} e'}{e, \sigma \mapsto e', \sigma.}$$

Next we extend the notion of **value** such that locations $l \in Loc$ are considered values, too:

$$v ::= \ldots \mid\ l.$$

We say that a program is *well-formed* if it does not contain any locations.

Accordingly, we extend the **evaluation contexts**:

$$C ::= \ldots \mid \mathsf{ref}\ C \mid !C \mid C := e \mid v := C \mid C; e.$$

The evaluation context definition enforces a strict left-to-right evaluation order on the := expression. This is important in order to retain the Church-Rosser property.

We are now able to write down the **SOS**:

$$\frac{l \notin dom(\sigma)}{ref v, \sigma \mapsto l, \sigma[l \mapsto v]} \qquad \frac{}{!l, \sigma \mapsto \sigma(l), \sigma}$$

$$\frac{}{l := v, \sigma \mapsto \#u, \sigma[l \mapsto v]} \qquad \frac{}{v; e, \sigma \mapsto e, \sigma.}$$

Notice that the first rule has a side condition $l \notin \mathsf{dom}(\sigma)$, ensuring that the newly allocated location l is not previously bound in the store σ.

We define the result of the := expression to be the unit value $\#u$ to reinforce the idea that this is an expression evaluated for its side-effect.

3 Translation to **uF**

Given an expression e in **uF!**, an environment ρ and a state σ, we define $\mathcal{D}[\![e]\!]\rho\sigma$ to be the **uF** term that evaluates e in ρ and σ to some value v and returns the pair $\langle v, \sigma' \rangle$, where σ' is the store after executing e. We will assume that the environment ρ and store σ are **uF** terms; initially, the environment is some ρ_0 and the state is σ_0, since it doesn't matter what they are until we want to check errors.

Before defining the translation, we introduce three functions we'll make use of:

- $\mathsf{malloc}\ \sigma = l$: returns the location l not allocated in σ
- $\mathsf{lookup}\ \sigma\ l = \sigma(l)$: returns the value stored at location l in state σ
- $\mathsf{update}\ \sigma\ l\ v = \sigma[l \mapsto v]$: the state is updated such that value v is stored at the location l.

There are many possible implementations of these operations; we require only that they satisfy the following specification (the operation allocated is needed to write the specification and to implement an error-checking version of the semantics):

$$\mathsf{lookup}(\mathsf{update}(s\ l\ v)\ l) = v$$

$$\mathsf{lookup}(\mathsf{update}(s\ l\ v)\ l') = \mathsf{lookup}(s\ l'),\ \text{where } l \neq l'$$

$$\mathsf{allocated}(\mathsf{malloc}(\sigma)\ \sigma) = \mathsf{false}$$

$$\mathsf{allocated}(l\ \mathsf{update}(\sigma\ l\ v)) = \mathsf{true}$$

$$\mathsf{allocated}(l\ \sigma_0) = \mathsf{false}$$

$$\mathsf{update}(\mathsf{update}(\sigma\ l\ v)\ l'\ v') = \mathsf{update}(\mathsf{update}(\sigma\ l'\ v')\ l\ v),\ \text{where } l \neq l'$$

$$\mathsf{update}(\mathsf{update}(\sigma\ l\ v)\ l\ v') = \mathsf{update}(\sigma\ l\ v').$$

We now give the translation:

(1) $\mathcal{D}[\![n]\!]\rho\sigma = \langle n, \sigma \rangle$

(2) $\mathcal{D}[\![x]\!]\rho\sigma = \langle \rho \text{“}x\text{“}, \sigma \rangle$

(3) $\mathcal{D}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho\sigma =$
$= \mathsf{let}\ p_0 = \mathcal{D}[\![e_0]\!]\rho\sigma\ \mathsf{in}$
$\mathsf{let}\ b = \mathsf{left}\ p_0\ \mathsf{in}$
$\mathsf{let}\ \sigma' = \mathsf{right}\ p_0\ \mathsf{in}$
$\mathsf{if}\ b\ \mathsf{then}\ \mathcal{D}[\![e_1]\!]\rho\sigma'\ \mathsf{else}\ \mathcal{D}[\![e_2]\!]\rho\sigma'$

(4)$\mathcal{D}[\![e_1; e_2]\!]\rho\sigma = \mathsf{let}\ p_1 = \mathcal{D}[\![e_1]\!]\rho\sigma\ \mathsf{in}$
$\mathsf{let}\ \sigma' = \mathsf{right}\ p_1\ \mathsf{in}$
$\mathcal{D}[\![e_2]\!]\rho\sigma'$

(5)$\mathcal{D}[\![\mathsf{ref}\ e]\!]\rho\sigma = \mathsf{let}\ p_0 = \mathcal{D}[\![e]\!]\rho\sigma\ \mathsf{in}$
$\mathsf{let}\ v = \mathsf{left}\ p_0\ \mathsf{in}$
$\mathsf{let}\ \sigma' = \mathsf{right}\ p_0\ \mathsf{in}$
$\mathsf{let}\ l = \mathsf{malloc}\ \sigma'\ \mathsf{in}$
$\langle l, \mathsf{update\_store}\ \sigma' l\ v \rangle$

(6)$\mathcal{D}[\![!e]\!]\rho\sigma = \mathsf{let}\ p_0 = \mathcal{D}[\![e]\!]\rho\sigma\ \mathsf{in}$
$\mathsf{let}\ v = \mathsf{left}\ p_0\ \mathsf{in}$
$\mathsf{let}\ \sigma' = \mathsf{right}\ p_0\ \mathsf{in}$
$\langle \mathsf{lookup}\ \sigma'\ v, \sigma' \rangle$

(7)$\mathcal{D}[\![e_1 := e_2]\!]\rho\sigma = \mathsf{let}\ p_1 = \mathcal{D}[\![e_1]\!]\rho\sigma\ \mathsf{in}$
$\mathsf{let}\ l = \mathsf{left}\ p_1\ \mathsf{in}$
$\sigma' = \mathsf{right}\ p_1\ \mathsf{in}$

$$
\begin{array}{l}
\mathsf{let}\ p_2 = \mathcal{D}[\![e_2]\!]\rho\sigma'\ \mathsf{in} \\
\quad \mathsf{let}\ v = \mathsf{left}\ p_2\ \mathsf{in} \\
\quad\quad \mathsf{let}\ \sigma'' = \mathsf{right}\ p_2\ \mathsf{in} \\
\quad\quad\quad \langle \#u, \mathsf{update\_store}\ \sigma''\ l\ v\rangle
\end{array}
$$

Some explanations are need. For example **(1)** should evaluate n in ρ and σ, which of course is n, and return the pair $\langle n, \sigma\rangle$; much in the same way, in **(2)** we should evaluate variable x in ρ and σ, which is ρ"x", and return it in pair with σ. The rest of the rules are recurrent: each time we take the translation of an expression in ρ and σ and get a pair from which we extract the actual value and the new state and then perform translations in the same environment ρ, but in the new state. Since the environment is not changed, these translation rules show the difference between environments and states.

We said that environments and states are treated as **uF** terms; to give an example, rule **(1)** may be rewritten as $\mathcal{D}[\![n]\!] = (\lambda\, \rho\, (\lambda\, \sigma\, \langle n, \sigma\rangle))$.

Not all the times we are interested in the actual value an expression evaluates to in ρ and σ; for example in rule **(4)** we only need to translate e_1 and then make explicit the new state, required for the translation of e_2.

We must also pay attention to all the possible side-effects: in rule **(5)** e may have side effects, such that we do not actually create a new location and assign a value to it in the state σ where e is evaluated, but in the state σ' resulted from the evaluation.

The *malloc* function should not be mistaken for the similar function in **C**, since it just returns a location not allocated in the current state, and no updates are done; successive calls to *malloc* return the same location.

Thinking about these rules, it becomes apparent that at any given point exactly one state is needed. So it is possible to have a single state, and having only one state at each time would avoid the problem of creating a large number of states. However, there are language features like transactions that require duplication of the state, semantically at least.

4 Mutable Variables

Suppose now that we want all variables to be mutable. We extend the **uF** expressions to

$$e ::= \ldots \mid x := e \mid e_1; e_2.$$

We can desugar this extended **uF** to **uF!** and let the translation of such an expression e to be $\mathcal{M}[\![e]\!]$ given by the following rules:

(1)$\mathcal{M}[\![x]\!] = !x$

(2)$\mathcal{M}[\![x := e]\!] = \ x := \mathcal{M}[\![e]\!]$

(3)$\mathcal{M}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] = \mathsf{let}\ x = \mathsf{ref}\ \mathcal{M}[\![e_1]\!]\ \mathsf{in}\ \mathcal{M}[\![e_2]\!]$

(4)$\mathcal{M}[\![\lambda x\ e]\!] = \lambda x\ \mathcal{M}[\![e]\!] = \lambda x'\ \mathsf{let}\ x = \mathsf{ref}\ x'\ \mathsf{in}\ \mathcal{M}[\![e]\!]$

(5)$\mathcal{M}[\![e_0\ e_1]\!] = \mathcal{M}[\![e_0]\!]\ \mathsf{ref}\ \mathcal{M}[\![e_1]\!])$.

Note: We have to make sure that all variables are assignable.