CS611 Lecture 12 — State-passing and mutable variables — 24 September, 2004
Scribe: Yejin Choi, Krzysztof Ostrowski — Lecturer: Andrew Myers

1 State-passing style translations

In the previous class we extended our untyped functional language **uML** with ML-style references to memory locations and described a structural operational semantics for this extension[1].

$$e ::= \; n \mid x \mid \mathsf{let}\ x \;=\; e_1\ \mathsf{in}\ e_2 \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{true} \mid \mathsf{false} \mid (e_1, ..., e_n) \mid \#n\ e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid ...$$

$$\boxed{... \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \mathsf{unit}}$$

Since the new features introduced *side-effects* to the language, we decided to extend our evaluation relation with a new component of *state* to take this factor into account. However, dealing with state explicitly as a context in which evaluation is made is not the only way to describe the behavior of a language with such features. In this class we learn how to translate[2] programs in the new language back into the pure **uML** by means of passing *state* as an argument to and a result of our evaluation function.

The new translation function maps syntactical objects representing expressions into functions that, given naming environment and a state (represented as a function from locations[3] to values) return a pair consisting of the value of the evaluated expression and a new state that reflects all the side effects.

$$[\![e]\!] = \lambda\rho.\lambda\sigma.(v, \sigma') \quad - \quad \text{evaluate } e \text{ in state } \sigma \text{ and environment } \rho\text{, return } v \text{ and a new state } \sigma'$$

In the literature this sort of technique is sometimes called a *state-passing style.*

As a side note, the language we are describing here is roughly equivalent to ML. The main difference is in that ML is typed; types, however, do not affect the execution, but rather serve merely as a means of checking of program correctness.

2 Rules for translation of uML! into uML

Translation of integers and variables is straightforward: they do not affect the state.

$$\begin{aligned} [\![n]\!]\rho\sigma &= (n, \sigma) \\ [\![x]\!]\rho\sigma &= (\rho``x", \sigma) \end{aligned}$$

During the evaluation of the if expression, the branching condition may have side effects, hence the need to pass the updated state to the appropriate subexpression.

$$[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho\sigma = \mathsf{let}\ p = [\![e_0]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ b = \#1\ p\ \mathsf{in}\ \mathsf{let}\ \sigma' = \#2\ p\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ [\![e_1]\!]\rho\sigma'\ \mathsf{else}\ [\![e_2]\!]\rho\sigma'$$

Since the new state and value are passed back as a pair, we needed to extract them first. For brevity, we now extend let with a new pattern-matching style notation and rewrite the above as the following.

$$[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho\sigma = \mathsf{let}\ (b, \sigma') = [\![e_0]\!]\rho\sigma\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ [\![e_1]\!]\rho\sigma'\ \mathsf{else}\ [\![e_2]\!]\rho\sigma'$$

We will use this notation throughout the lecture.

The above illustrates the mechanics of our translation, handling tuples, indexing etc. is similar. We now turn to describing the semantics of the newly introduced programming constructs.

[1]Note that we simplified uML a little bit, omitting multiargument functions and letrec. This is for brevity and simplicity, they would not introduce anything new beyond what is presented here.

[2]We rely on the fact that the semantics of **uML** has already been described via translating it into the lambda calculus, hence showing how to translate **uML!** into **uML** is enough to specify the behavior of programs in **uML!**.

[3]How exactly locations are represented is an implementation detail, we may assume e.g. that locations are integer numbers.

The expression $\mathsf{ref}\ e$ stores the value of e in some newly allocated location and returns that location, together with an updated state, as a result. We model this as a two-step process via a pair of semantic functions, $\mathsf{malloc} : (\mathsf{Loc}^{4} \to \mathsf{Val}) \to \mathsf{Loc} \times (\mathsf{Loc} \to \mathsf{Val})$, which returns a new, unused location and a modified state where this location is no longer free, and $\mathsf{update} : (\mathsf{Loc} \to \mathsf{Val}) \times \mathsf{Loc} \times \mathsf{Val} \to (\mathsf{Loc} \to \mathsf{Val})$, which updates a given state by putting a specified value under a specified location (and again returns the resulting state). Formally, we guarantee that if $\mathsf{malloc}\ \sigma = (l, \sigma')$ then $l \notin \mathsf{dom}(\sigma)$ and that $\mathsf{update}\ \sigma\ l\ v = \sigma[l \to v]$.

$$[\![\mathsf{ref}\ e]\!]\rho\sigma = \mathsf{let}\ (v, \sigma') = [\![e]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (l, \sigma'') = \mathsf{malloc}\ \sigma'\ \mathsf{in}\ (l, \mathsf{update}\ \sigma''\ l\ v)$$

Expression $!e$ retrieves a value stored under a location represented by e. We again introduce a new semantic function, $\mathsf{lookup} : (\mathsf{Loc} \to \mathsf{Val}) \times \mathsf{Loc} \to \mathsf{Val}$ that does the job. Notice that we lookup the value from the new state, in case the evaluation of e altered the content at the location specified e itself.

$$[\![!e]\!]\rho\sigma = \mathsf{let}\ (l, \sigma') = [\![e]\!]\rho\sigma\ \mathsf{in}\ (\mathsf{lookup}\ \sigma'\ l, \sigma')$$

We don't really care how the functions lookup, update are implemented as long as they satisfy the following *equational specification*:

$$\begin{array}{rcl}
\mathsf{lookup}(\mathsf{update}\ \sigma\ l\ v)\ l & = & v \\
\mathsf{lookup}(\mathsf{update}\ \sigma\ l\ v)\ l' & = & \mathsf{lookup}\ \sigma\ l' \qquad \text{for}\ \ l \neq l' \\
\mathsf{update}(\mathsf{update}\ \sigma\ l\ v)\ l\ v') & = & \mathsf{update}\ \sigma\ l\ v' \\
\mathsf{update}(\mathsf{update}\ \sigma\ l\ v)\ l'\ v') & = & \mathsf{update}\ (\mathsf{update}\ \sigma\ l'\ v')\ l\ v \qquad \text{for}\ \ l \neq l' \\
(\mathsf{malloc}\ \sigma) & = & (l, \sigma') \\
 & & \text{where}\quad (\mathsf{allocated}\ \sigma'\ l)\ \wedge\ \neg(\mathsf{allocated}\ \sigma\ l) \\
 & & \qquad \wedge\ (\forall l'\ .\ (\mathsf{allocated}\ \sigma\ l') \Rightarrow \mathsf{lookup}\ \sigma\ l' = \mathsf{lookup}\ \sigma'\ l') \\
(\mathsf{allocated}\ \sigma_0\ l) & = & \mathit{false} \qquad \text{where}\ \sigma_0\ \text{is the initial memory}
\end{array}$$

This specification doesn't dictate a representation of locations or of states, but the equations given allow us to prove anything we would want, to by substituting equals for equals. For example, we might represent locations as integers, and a state as a pair (f, n) where $f(2)$ gives the value at location 2. The number n represents a location above which all locations are unallocated, so we'd have

$$\begin{array}{rcl}
\mathsf{malloc}\ (f, n) & = & (n, (f, n+1)) \\
\mathsf{allocated}\ (f, n)\ m & = & (m < n) \\
\mathsf{lookup}\ (f, n)\ m & = & f(m) \\
\mathsf{update}\ (f, n)\ m\ v & = & (\lambda m'.\, \mathbf{if}\ m = m'\ \mathbf{then}\ v\ \mathbf{else}\ f(m'), n) \\
\sigma_0 & = & (\lambda m.\, 0, 0)
\end{array}$$

But this is just one possible implementation. If we wanted automatic garbage collection, we'd need a more complicated representation of states.

In translating assignment, we need to decide about the order in which expressions are evaluated, we assume a left-to-right evaluation, i.e., the location is calculated first, hence the value to be assigned is evaluated in the state affected by the evaluation of the location. Note that the returned value is unit, this is to stress the fact that the primary purpose of using the assignment is for its side-effect.

$$[\![e_1 := e_2]\!]\rho\sigma = \qquad \mathsf{let}\ (l, \sigma') = [\![e_1]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (v, \sigma'') = [\![e_2]\!]\rho\sigma'\ \mathsf{in}\ (\mathsf{unit}, \mathsf{update}\ \sigma''\ l\ v)$$

So much for the new features. We might expect that the translation of lambda expression and application would be similar to what we have seen for the other parts of the old langague, hence we might be tempted to write the following, with update_env being a function updating the environment accordingly.

[4]Here Loc and Val represent the sets of locations and values, respectively. According to our earlier definition, the state is a mapping from locations to values, hence it is of type $\mathsf{Loc} \to \mathsf{Val}$.

$$[\![\lambda x.e]\!]\rho\sigma = \qquad (\lambda y.[\![e]\!](\mathsf{update\_env}\ \rho\ \text{“}x\text{”}\ y)\sigma, \sigma)$$

The problem with such a definition is that it would cause the function to evaluate its body in the state from the time when lambda expression was introduced and assigned to a variable or stored in memory, not at the time when it is being applied. This makes us realize that a function needs to take an additional argument, the current state at the time when it is being applied, and evaluate its body in this state. Note that even though we are passing the state in a *dynamic* fashion, this semantic is still *static scoping*, as static and dynamic scoping are regarding the naming environment(the fuction to map names of variables to locations) and here we are using the naming environment at the definition time of the function.

$$[\![\lambda x.e]\!]\rho\sigma = \qquad (\lambda y.\lambda\sigma'.[\![e]\!](\mathsf{update\_env}\ \rho\ \text{“}x\text{”}\ y)\sigma', \sigma)$$

When applying the function to an argument, we first evaluate the function, then evaluate the expression in the state affected by the previous evaluation, and finally pass the resulting state together with the value to the function for processing.

$$[\![e_1\ e_2]\!]\rho\sigma = \qquad \mathsf{let}\ (f, \sigma') = [\![e_1]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (v, \sigma'') = [\![e_2]\!]\rho\sigma'\ \mathsf{in}\ f\ v\ \sigma''$$

3 Implementation considerations

Historically, lazy languages didn't have states, because the notion of *state* does not make much of sense in lazy language. That is, when there is an expression whose evaluation has not been performed, the value of such evaluation is not well defined.

Another thing to notice is that the implementation of states can be done quite efficiently. This is because at any given moment of execution, there is only one state, and whenever the side effects of the program alters the state, the old state is simply overwritten with the new one. That is, we can implement the state by *destructive update*.

With transactions or checkpointing mechanisms, we will need to keep multiple states floating around, in case we need to fall back to a state in the past.

4 Mutable variables

The expressions that we've examined so far were based on ML. In this section, we will show how we can describe the semantics of C like language features, such as pointer and reference, by means of translation rules. We will extend our simple language to include the following expressions.

$$\boxed{...\mid e_1 = e_2 \mid \&e \mid *e}$$

Here the assignment puts the value of expression e_2 into the location represented by e_1 and returns the new value at the location of e_1. $\&e$ returns the location of e, and $*e$ returns the value stored at location e.

Note that expressions in this language can be interpreted in two different ways depending on the context in which they are used. An expression at the left side of an assignment operator represents a *location* of that expression, whereas an expression on the right side represents its *value*. In order to stress this dependency we define *lvalues* as terms that may appear on the left hand side of an assightment, and *rvalues* as terms that may not appear on the left hand side of an assightment. Also, we will introduce two corresponding translation functions, $\mathcal{L}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]$, accordingly.

Translations of numbers and variables are straightforward. The former can never act as *lvalues*, therefore no corresponding translation is shown.

$$\mathcal{R}[\![n]\!]\rho\sigma = (n, \sigma)$$

$$\mathcal{R}[\![x]\!]\rho\sigma = (\mathsf{lookup}\ \sigma\ (\rho\text{“}x\text{”}), \sigma)$$

$$\mathcal{L}[\![x]\!]\rho\sigma = (\rho\text{“}x\text{”}, \sigma)$$

Note that in general, if $\mathcal{L}[\![e]\!]$ is defined (and, as we have just seen, it not always is), then $\mathcal{R}[\![e]\!]$ will always be expressed simply as a lookup into the corresponding memory location, hence the following principle.

$$\mathcal{R}[\![e]\!]\rho\sigma = \mathsf{let}\ (l, \sigma') = \mathcal{L}[\![e]\!]\rho\sigma\ \mathsf{in}\ (\mathsf{lookup}\ \sigma'\ l, \sigma')$$

.

Since variables in this language represent memory locations, translation of the let expression needs to be slightly modified, it is now going to be roughly equivalent to $\mathcal{R}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma \cong \mathsf{let}\ x = \ \mathsf{ref}\ e_1\ \mathsf{in}\ e_2$ in **uML$_!$**, that is, assigning a reference.

$$\mathcal{R}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma = \mathsf{let}\ (v, \sigma') = \mathcal{R}[\![e_1]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (l, \sigma'') = \ \mathsf{malloc}\ \sigma'\ \mathsf{in}\ \mathcal{R}[\![e_2]\!](\rho[\text{“}x\text{”} \mapsto l])(\mathsf{update}\ \sigma''\ l\ v)$$

Assignment is almost identical as in **uML$_!$**, but we need to apply the appropriate translation functions on the two sides of the assignment operator. Also note that here the assigntment will return the value of the assigned value, as in C.

$$\mathcal{R}[\![e_1 = e_2]\!]\rho\sigma = \mathsf{let}\ (l, \sigma') = \mathcal{L}[\![e_1]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (v, \sigma'') = \mathcal{R}[\![e_2]\!]\rho\sigma'\ \mathsf{in}\ (v, \mathsf{update}\ \sigma''\ l\ v)$$

Finally, the & and $*$ operators. Note that the former can never appear as a *lvalue* while the latter can.

$$\mathcal{R}[\![\&e]\!]\rho\sigma = \mathcal{L}[\![e]\!]\rho\sigma$$

$$\mathcal{L}[\![*e]\!]\rho\sigma = \mathcal{R}[\![e]\!]\rho\sigma$$

We also must modify functions because the formal argument of a variable is now allocated in the store:

$$\begin{aligned}
\mathcal{R}[\![\lambda x.\, e]\!]\rho\sigma &= \lambda y.\, \lambda\sigma'.\, \mathsf{let}\ (l, \sigma') = \mathsf{malloc}\ \sigma\ \mathsf{in}\ \mathsf{let}\ \sigma'' = (\mathsf{update}\ \sigma'\ l\ y)\ \mathsf{in}\ \mathcal{R}[\![e]\!](\rho[\text{“x”} \mapsto y])\sigma'' \\
\mathcal{R}[\![e_0\ e_1]\!] &= \mathsf{let}\ (f, \sigma') = \mathcal{R}[\![e_1]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (v, \sigma'') = \mathcal{R}[\![e_2]\!]\rho\sigma'\ \mathsf{in}\ f v \sigma''
\end{aligned}$$

5 Call by reference

Some languages, such as Pascal and Ada, support a parameter-passing style in which an actual parameter being passed to a function can be updated by that function. That is, the actual variable is passed to a function, rather than the value in the variable. This feature can be easily modeled by translation. We add a new kind of function and a new kind of application:

$$e ::= \ldots \mid \lambda_r x\,.\,e \mid \mathsf{rapp}\ e_0\ e_1$$

Here is a translation of these new features. The key idea is that we no longer need to allocate a new variable in the function, and the application passes the *lvalue* of the argument.

$$\begin{aligned}
\mathcal{R}[\![\lambda_r x.\, e]\!]\rho\sigma &= \lambda l.\, \lambda\sigma'.\, \mathcal{R}[\![e]\!]\rho[\text{“x”} \mapsto l]\sigma' \\
\mathcal{R}[\![\mathsf{rapp}\ e_0\ e_1]\!] &= \mathsf{let}\ (f, \sigma') = \mathcal{R}[\![e_0]\!]\rho\sigma\ \mathsf{in}\ \mathsf{let}\ (l, \sigma'') = \mathcal{L}[\![e_1]\!]\rho\sigma'\ \mathsf{in}\ f l \sigma''
\end{aligned}$$