Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

## 1 Some thoughts about state

State refers to the ability to have mutation: a change in value over time. The functional languages we have studied so far, such as the $\lambda$-calculus and uML, have not had state, in the sense that once a value is created, it is impossible to change that value, and once a variable is bound it cannot be rebound to a new value (except by creating a new variable). Although state is not a necessary feature of a programming language (e.g. $\lambda$-calculus is Turing complete but does not have a notion of state), it is a feature of most languages in common use today. We distinguish between functional languages and imperative languages primarily on the basis of whether they are organized primarily around a stateless paradigm (with higher-order functions), or an stateful paradigm of mutable variables and data structures. It might be more useful, in fact, to distinguish between *stateless* and *stateful* languages, since stateful languages usually have functions too.

Why do many programmers like stateful programming languages? Perhaps because it matches how we think of the world. We view the world as being composed of entities (such as people or objects) with intrinsic unchanging identities. These entities change their state over time while maintaining the same underlying identity. If a programming language matches our mental model of the world, it's more comfortable for programmers to think about.

Thus, stateful languages always come equipped with an underlying notion of *identity*, such as variable identity or object identity. In these languages, the binding between identities and their current states can change over time. Stateful languages intrinsically have a notion of time, unlike pure functional languages such as uML or lambda calculus, which take an essentially timeless view of computation. In stateless languages, it doesn't matter exactly when something happens, because the outcome will be the same regardless. In stateful languages with multiple concurrent threads, it matters a great deal when something happens.

It can be argued that our perception of underlying identity is an illusion. In physics, identity does not exist: particles are identified by their state, not by any intrinsic identity. If two electrons in different states are swapped, no change has happened to the universe, because an electron is really the same thing as its state. Time is arguably an illusion as well: the universe is a multidimensional object in which physics says that time is much more symmetric to the other dimensions than we perceive. Being made of particles ourselves, we are only able to sense a single moment in time. Arguably, the universe is purely functional, and our minds deceive us. In fact, the observation that we are deceived into thinking that objects have an underlying identity dates back to Buddhist and even earlier Vedantic philosophy (the Upanishads).

The mismatch between reality and our perceptions becomes pragmatically relevant when we look at the difficulties in programming multiprocessor/multicore systems. Our consciousness is single-threaded, so it makes perfect sense to us to think about state evolving along a single timeline. But when with multiple simultaneous threads of computation, and communication delays, it is infeasible to give all threads a single, consistent view of the state evolution of the system. We find it very difficult to reason about how such multithreaded systems work and to write correct programs for them in a stateful language.

Designing workable programming languages for multicore systems is currently an active research topic. In sidestepping the intrinsic problems with state, a stateless language may be a better match for such systems than the stateful languages we commonly use!

## 2 First-class references

We extend uML with state, and call the new language uML$_!$. We use a construct called a *reference cell* that allows a value to be rebound to different values over time. We can then use reference cells to model language features such as mutable variables. Reference cells are first-class values in this language, like in ML. The syntax of uML$_!$ is as follows:

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \mathbf{ref}\ e & \text{(create a reference cell with initial value from } e\text{)} \\ & | & !e & \text{(evaluation or dereference)} \\ & | & e_1\ :=\ e_2 & \text{(assignment or mutation)} \\ & | & e_1;\ e_2 & \text{(sequential composition)} \end{array}$$

Informally, the meaning of these new expressions is as follows. Evaluating the expression **ref** $e$ creates a new reference cell having the initial value $e$. The expression $!e$ evaluates to the current value of the reference cell $e$. The assignment $e_1 := e_2$ assigns the value of $e_2$ to the reference cell $e_1$, and in our semantics returns the value **null**.

Other design choices could have been made. For example, the assignment operator could have returned the value of $e_2$ so that expressions like $e_0\ :=\ (e_1\ :=\ e_2)$ would make sense and could be used to assign the same value to multiple variables.

In this treatment, reference cells are first-class objects. This is more powerful than simple mutable variables as in C. It is also more dangerous, because it introduces *aliasing*, as the following example demonstrates:

$$\begin{array}{l} \mathbf{let}\ x = \mathbf{ref}\ 1\ \mathbf{in} \\ \quad \mathbf{let}\ y = x\ \mathbf{in} \\ \quad\quad \mathbf{let}\ z = (x := 2)\ \mathbf{in} \\ \quad\quad\quad !y \end{array}$$

Here $y$ is an alias for $x$. Dereferencing $y$ gives us the value of $x$, which in this case has changed from 1 to 2. The expression therefore evaluates to 2. In other words, if you kick $x$, $y$ jumps!

Aliasing is a constant source of errors in stateful programming. A classic example is a procedure that is supposed to multiply two matrices $a$ and $b$ and write the result imperatively into a third matrix $c$. The programmer writes the obvious code, then later wants to set $a$ to the product of $a$ and $b$. This makes the matrix multiply compute something completely different from what was intended, and is very hard to debug!

### 2.1 Operational semantics

We now give an operational semantics for reference cell expressions in uML$_!$. To do this, we define a *configuration* as a pair $\langle e, \sigma \rangle$, where $e$ is an expression and $\sigma$ is a function mapping reference cells to values. Reference cells are denoted generically by $\ell$. We extend the grammar of expressions and values to include reference cells.

$$\begin{array}{rcl} e & ::= & \ldots\ |\ \ell \\ v & ::= & \ldots\ |\ \ell \end{array}$$

We inherit the existing reductions from uML, lifting them to apply to uML$_!$ expressions. All these reductions have no side effects, so the state $\sigma$ is unchanged by them:

$$\frac{e\ \longrightarrow e'}{\langle e, \sigma \rangle\ \underset{\text{uML}_!}{\longrightarrow}\ \langle e', \sigma \rangle}$$

$$\begin{array}{rll} \langle \mathbf{ref}\ v, \sigma \rangle & \longrightarrow \langle \ell, \sigma[\ell \mapsto v] \rangle & \text{(where } \ell \notin \text{dom}(\sigma)\text{)} \\ \langle \ell := v, \sigma \rangle & \longrightarrow \langle \mathbf{null}, \sigma[\ell \mapsto v] \rangle & \text{(where } \ell \in \text{dom}(\sigma)\text{)} \\ \langle !\ell, \sigma \rangle & \longrightarrow \langle \sigma(\ell), \sigma \rangle & \text{(where } \ell \in \text{dom}(\sigma)\text{)} \\ \langle \mathbf{null};\ e, \sigma \rangle & \longrightarrow \langle e, \sigma \rangle & \end{array}$$

where $\sigma[v \mapsto \ell]$ is the same function as $\sigma$, except that the value on input $\ell$ is changed to $v$. The language has eager, left-to-right evaluation, so the evaluation contexts are given by this grammar:

$$E \quad ::= \quad \ldots \;\mid\; \mathbf{ref}\, E \;\mid\; !\,E \;\mid\; E := e \;\mid\; \ell := E \;\mid\; E; e$$

The structural congruence rule is adapted to take into account the current state:

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle}{\langle E[\,e\,], \sigma \rangle \longrightarrow \langle E[\,e'\,], \sigma' \rangle}$$

## 2.2 Translation semantics

$[\![e]\!]\rho\sigma$ is an expression that computes a pair $(e', \sigma')$ in a naming environment $\rho$ and state $\sigma$, where $e'$ is the resulting expression and $\sigma'$ is the store.

$$\begin{array}{rcl}
[\![n]\!]\rho\sigma & = & (n, \sigma) \\
[\![x]\!]\rho\sigma & = & (\rho \text{ ``}x\text{''}, \sigma) \\
[\![\mathbf{if}\ e_b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\rho\sigma & = & \mathbf{let}\ (b, \sigma')^1 = [\![e_b]\!]\rho\sigma\ \mathbf{in} \\
 & & \mathbf{if}\ b\ \mathbf{then}\ [\![e_1]\!]\rho\sigma'\ \mathbf{else}\ [\![e_2]\!]\rho\sigma' \\
[\![\mathbf{ref}\ e]\!]\rho\sigma & = & \mathbf{let}\ (x, \sigma') = [\![e]\!]\rho\sigma\ \mathbf{in}\ \mathbf{let}\ l = \mathit{MALLOC}\ \sigma'\ x\ \mathbf{in}\ (l, \mathit{UPDATE}\ \sigma'\ l\ x) \\
[\![\,!e]\!]\rho\sigma & = & \mathbf{let}\ (x, \sigma') = [\![e]\!]\rho\sigma\ \mathbf{in}\ (\mathit{LOOKUP}\ \sigma'\ x, \sigma') \\
[\![e_1\ :=\ e_2]\!]\rho\sigma & = & \mathbf{let}\ (x_1, \sigma_1) = [\![e_1]\!]\rho\sigma\ \mathbf{in} \\
 & & \mathbf{let}\ (x_2, \sigma_2) = [\![e_2]\!]\rho\sigma_1\ \mathbf{in} \\
 & & (\mathbf{null},\ \mathit{UPDATE}\ \sigma_2\ x_1\ x_2) \\
[\![e_1; e_2]\!]\rho\sigma & = & \mathbf{let}\ (x, \sigma_1) = [\![e_1]\!]\rho\sigma\ \mathbf{in}\ [\![e_2]\!]\rho\sigma_1 \\
[\![\mathbf{null}]\!]\rho\sigma & = & (\mathbf{null}, \sigma) \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]\rho\sigma & = & \mathbf{let}\ (y, \sigma') = [\![e_1]\!]\rho\sigma\ \mathbf{in}\ [\![e_2]\!](\mathit{EXTEND}\ \rho\ \text{``}x\text{''}\ y)\sigma' \\
[\![\lambda x.\, e]\!]\rho_{lex}\sigma_{lex} & = & \lambda y\sigma_{dyn}.\, [\![e]\!]\ (\mathit{EXTEND}\ \rho_{lex}\ \text{``}x\text{''}\ y)\ \sigma_{dyn} \\
[\![e_1\ e_2]\!]\rho_{dyn}\sigma_{dyn} & = & \mathbf{let}\ (f, \sigma_1) = [\![e_1]\!]\rho_{dyn}\sigma_{dyn}\ \mathbf{in} \\
 & & \mathbf{let}\ (v, \sigma_2) = [\![e_2]\!]\rho_{dyn}\sigma_1\ \mathbf{in} \\
 & & f\ v\ \sigma_2
\end{array}$$

Here, we rely on certain abbreviations (*MALLOC*, *UPDATE*, and *LOOKUP*), for manipulating stores. The function *MALLOC* allocates a new memory location; *LOOKUP* reads the contents of a memory location; *UPDATE* produces a new store with one location changed. Further, we will need an initial store *EMPTY-STORE*. We don't care exactly how these terms are implemented, as long as they obey certain equations (the notion of equality here is equivalence after evaluation):

$$\begin{array}{rcll}
\mathit{LOOKUP}\ \mathit{EMPTY\text{-}STORE}\ \ell & = & \mathbf{error} & \\
\mathit{LOOKUP}\ \sigma\ (\mathit{MALLOC}\ \sigma\ v) & = & \mathbf{error} & \\
\mathit{LOOKUP}\ (\mathit{UPDATE}\ \sigma\ \ell\ v)\ \ell & = & v & \\
\mathit{LOOKUP}\ (\mathit{UPDATE}\ \sigma\ \ell\ v)\ \ell' & = & \mathit{LOOKUP}\ \sigma\ \ell' & (\text{if } \ell \neq \ell')
\end{array}$$

Implementing terms with these properties is left as an exercise to the reader.

[1] '$\mathbf{let}\ (b, \sigma') = [\![e_b]\!]\rho\sigma\ \mathbf{in}\ e$' here is syntactic sugar for $\mathbf{let}\ p = [\![e_b]\!]\rho\sigma\ \mathbf{in}\ \mathbf{let}\ b = \#1\ p\ \mathbf{in}\ \mathbf{let}\ \sigma' = \#2\ p\ \mathbf{in}\ e$'.

## 3 Mutable variables

Let's take a closer look at what happens in a language like C, where variables themselves are mutable.

### 3.1 Syntax

$$e \quad ::= \quad \ldots\text{uML}\ldots \;\mid\; e_1 = e_2 \;\mid\; *\,e \;\mid\; \&e \;\mid\; e_1; e_2$$

where we think of the constructs respectively as:

- $e_1 = e_2$ is assignment where $e_1$ is an assignable variable name, for instance, an array reference.
- $*e$ is used to dereference a pointer $e$ (acts sort of like ! in UML$_!$).
- $\&e$ gives a pointer to the location of $e$.

### 3.2 Translation semantics

The target language is again uML. Since variables are mutable, we need to change the naming environment $\rho$ to be a mapping from variable names to store locations. Each variable is bound to the identity of its reference cell.

The translation we define takes into account the fact that C has two kinds of values:

- lvalues, values which can appear in the left-hand side of an assignment; and
- rvalues, values that cannot appear on the LHS.

To capture this in the target language, for each expression $e$ of C, we define $\mathcal{L}[\![e]\!]\rho\sigma$ (the lvalue of $e$) *and* $\mathcal{R}[\![e]\!]\rho\sigma$ (the rvalue of $e$). As before, we need to keep track of the state of the computation so the results of the translations are pairs. To summarize:

- $\mathcal{L}[\![e]\!]\rho\sigma = (\text{location of } e, \sigma')$
- $\mathcal{R}[\![e]\!]\rho\sigma = (\text{value of } e, \sigma')$.

The translation is defined inductively on the structure of C programs. Note that some lvalue translations are not defined, corresponding to invalid expressions that a C compiler would catch. For instance, $\mathcal{L}[\![n]\!]\rho\sigma$ is not defined. This prevents reassigning of, say, the integer 5 to have the value 3.

$$\begin{aligned}
\mathcal{R}[\![n]\!]\rho\sigma &= (n, \sigma) \\
\mathcal{L}[\![x]\!]\rho\sigma &= (\rho \text{ ``}x\text{''}, \sigma) \\
\mathcal{R}[\![e]\!]\rho\sigma &= \textbf{let}\,(\ell, \sigma') = \mathcal{L}[\![e]\!]\rho\sigma \textbf{ in } (\textit{LOOKUP}\ \sigma'\ \ell, \sigma') \qquad (\text{if } \mathcal{L}[\![e]\!] \text{ is defined})
\end{aligned}$$

For assignment:

$$\begin{aligned}
\mathcal{R}[\![e_1 = e_2]\!]\rho\sigma \quad = \quad & \textbf{let}(\ell, \sigma') = \mathcal{L}[\![e_1]\!]\rho\sigma\,\textbf{in} \\
& \textbf{let}(v, \sigma'') = \mathcal{R}[\![e_2]\!]\rho\sigma'\,\textbf{in}\ (v, \textit{UPDATE}\ \sigma''\ \ell\ v)
\end{aligned}$$

Comments:

- This forces left-to-right evaluation.
- It allows a chain of assignments. For example, $e_1 = e_2 = e_3 = e_4 = 5$ will assign 5 to each $e_i$ in our translation, just as in C.

- An assignment statement is not an lvalue in C, nor is it here.

Addressing:

$$\begin{aligned}\mathcal{L}[\![*e]\!]\rho\sigma &= \mathcal{R}[\![e]\!]\rho\sigma \\ \mathcal{R}[\![\&e]\!]\rho\sigma &= \mathcal{L}[\![e]\!]\rho\sigma\end{aligned}$$

So the two operations, $*, \&$, are inverses of each other; their function is to shift the view of a value rather than to do computation. For example, the semantics defined above let us write $*\&*\&*\&x = 2$. This will have the same effect as $x = 2$. Another example (written in C):

**int y = 0;**
**int x = &y;**
***x = 1;**

This piece of code assigns 1 to $y$. Note that it would be illegal to include a line **&y = 0x1002;** because we can't change the address of a variable.