State ISWIM

Alternative Formulation

CS 6520, Spring 2000

We've seen how state, as it's often used in imperative languages,

```
int sum(int n) {
  int i = 0;
  while (n > 0) {
    i = i + n;
    n = n - 1;
  }
  return n;
}
```

can be implemented with accumulating arguments in a purely functional setting:

```
(define (sum n i)
  (if (zero? n)
      i
      (sum (- n 1) (+ i n))))
```

But state and imperitive constructs have uses other than loops. For example, state is central to object-oriented programming. Let's extend our ISWIM model to accomodate state more directly. We can use some intuition from our encoding of state in loops: just as the state in `i` from the C program was emulated above by passing an `i` Scheme variable around, we'll think of state during evaluation as something that gets passed around in the evaluator.

State ISWIM

We extend the syntax of ISWIM expressions with a assignment expression:

$$M \quad := \quad \ldots \mid (:=\ X\ M)$$

Our intent is that assignment mutates a variable, so that later references to the variable get the new value. Evaluation will proceed roughly like the following sketch:

$$\begin{array}{lll}
 & ((\lambda x.((\lambda y.x)\ (:=\ x\ (+\ x\ 1))))\ 12) & \\
\longrightarrow & ((\lambda y.x)\ (:=\ x\ (+\ x\ 1))) & \text{with} \quad x = 12 \\
\longrightarrow & ((\lambda y.x)\ (:=\ x\ (+\ 12\ 1))) & \text{with} \quad x = 12 \\
\longrightarrow & ((\lambda y.x)\ (:=\ x\ 13)) & \text{with} \quad x = 12 \\
\longrightarrow & ((\lambda y.x)\ \mathsf{void}) & \text{with} \quad x = 13 \\
\longrightarrow & x & \text{with} \quad x = 13 \\
\longrightarrow & 13 &
\end{array}$$

From the above example, we get two clues about how to model state:

- We cannot use simple substitution when applying a function to an argument. For example, if we replaced every x with 12, we would end up with an ill-formed expression like $(:=\ 12\ (+\ 12\ 1))$, and we wouldn't know which 12s to replace with 13s later.

- Instead of rewriting an expression to an expression on each evluation step, we rewrote a pair on each step: an expression and a mapping from variables to values.

We might try to model := using an expression-envrionment pair, but our experience with CEK suggests that we shouldn't try to think of the variable mapping as an environment. For example, in CEK, we had to worry about the shadowing of variables in environments. Also, to deal with closures, we made a textual "copy" of the environment in each closure. Such copies would defeat the central mapping in the above example that made the := work, because $(\lambda x.y)$ would have its own separate environment.

The solution is to think of a mutable variable as a slot in the evaluator's memory. Function application will select an unused slot in memory, and replace all instances of the variable with the slot address. Then, variable lookup is replaced by slot lookup, and assignment corresponds to changing the value of a slot. The collection of slots is called a *store*.

To refine our above sketch, we'll rewrite expression-store pairs:

$$
\begin{array}{lll}
 & \langle\ ((\lambda x.((\lambda y.x)\ (:=\ x\ (+\ x\ 1))))\ 12) & ,\ \{\} \qquad\qquad \rangle \\
\longrightarrow & \langle\ ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ (+\ \sigma_1\ 1))) & ,\ \{\sigma_1 = 12\}\ \rangle \\
\longrightarrow & \langle\ ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ (+\ 12\ 1))) & ,\ \{\sigma_1 = 12\}\ \rangle \\
\longrightarrow & \langle\ ((\lambda y.\sigma_1)\ (:=\ \sigma_1\ 13)) & ,\ \{\sigma_1 = 12\}\ \rangle \\
\longrightarrow & \langle\ ((\lambda y.\sigma_1)\ \mathsf{void}) & ,\ \{\sigma_1 = 13\}\ \rangle \\
\longrightarrow & \langle\ \sigma_1 & ,\ \{\sigma_1 = 13\}\ \rangle \\
\longrightarrow & \langle\ 13 & ,\ \{\sigma_1 = 13\}\ \rangle
\end{array}
$$

With this approach, we need to extend ISWIM again to allow store identifiers as expressions and to support the void value:

$$
\begin{array}{lcl}
M & := & \ldots \mid (:=\ x\ M) \mid \sigma \\
V & := & b \mid (\lambda x.M) \mid \mathsf{void} \\
C & := & \ldots \mid (:=\ x\ C) \\
E & := & \ldots \mid (:=\ x\ E)
\end{array}
$$

Note that a store identifier is *not* a value, which means that when a variable (replaced by a store identifier) is used as an argument to a function, we must extract the value from the store before applying the function. That's the behavior we want for call-by-value (as opposed to call-by-reference).

Our reduction rules must now transform expression-store pairs, $\langle M, \Sigma \rangle$:

$$
\begin{array}{lcll}
\langle((\lambda x.M)\ V), \Sigma\rangle & \xrightarrow{s} & \langle M[\sigma/x], \Sigma[\sigma \leftarrow V]\rangle & \text{replaces the } \beta_v \text{ rule} \\
 & & \quad \text{where } \sigma \notin \mathrm{dom}(\Sigma) & \\
\langle \sigma, \Sigma\rangle & \xrightarrow{s} & \langle V, \Sigma\rangle & \\
 & & \quad \text{where } V = \Sigma(\sigma) & \\
\langle(:=\ \sigma\ V), \Sigma[\sigma \leftarrow V]'\rangle & \xrightarrow{s} & \langle \mathsf{void}, \Sigma[\sigma \leftarrow V]\rangle &
\end{array}
$$

Examples

Let's introduce some syntactic sugar to make State ISWIM examples easier to read:

$$
\begin{array}{lcl}
(\mathbf{let}\ x = M\ \mathbf{in}\ N) & = & ((\lambda x.N)\ M) \\
(\mathbf{let}\ x_1 = M_1\ \ldots\ x_n = M_n\ \mathbf{in}\ N) & = & (\mathbf{let}\ x_1 = M_1\ \mathbf{in}\ (\mathbf{let}\ x_2 = M_2\ \ldots\ x_n = M_n\ \mathbf{in}\ N)) \\
(\mathbf{begin}\ M_1\ \ldots\ M_n\ N) & = & (\mathbf{let}\ x_1 = M_1\ \ldots\ x_n = M_n\ \mathbf{in}\ N) \\
 & & \qquad \text{where } x_1, \ldots x_n \notin FV(N) \cup FV(M_1) \cup \ldots FV(M_n)
\end{array}
$$

With these abbreviations, we can write the original example as

$$
(\mathbf{let}\ x = 12\ \mathbf{in}\ (\mathbf{begin}\ (:=\ x\ 13)\ x))
$$

Here's another example:

$$
\begin{array}{l}
\langle\ \mathbf{let}\ i = 0 \qquad\qquad\qquad\qquad\qquad\quad ,\{\}\rangle \\
\qquad f = \mathbf{Y}\lambda f.\lambda x \\
\qquad\qquad \mathbf{if0}\ x \\
\qquad\qquad\quad i \\
\qquad\qquad\quad (\mathbf{begin} \\
\qquad\qquad\quad\ (:=\ i\ (+\ i\ x)) \\
\qquad\qquad\quad\ (f\ (-\ x\ 1))) \\
\ \ \mathbf{in} \\
\qquad (f\ 1)
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ \mathbf{let}\ f = \mathbf{Y}\lambda f.\lambda x \qquad\qquad\qquad\qquad ,\{\sigma_1 = 0\}\rangle \\
& \qquad\qquad \mathbf{if0}\ x \\
& \qquad\qquad\quad \sigma_1 \\
& \qquad\qquad\quad (\mathbf{begin} \\
& \qquad\qquad\quad\ (:=\ \sigma_1\ (+\ \sigma_1\ x)) \\
& \qquad\qquad\quad\ (f\ (-\ x\ 1))) \\
& \ \ \mathbf{in} \\
& \qquad (f\ 1)
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle(\sigma_2\ 1)\ ,\{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F\}\rangle
\end{array}
\qquad\qquad
\begin{array}{l}
\text{where } F = \lambda f.\lambda x \\
\qquad\qquad \mathbf{if0}\ x \\
\qquad\qquad\quad \sigma_1 \\
\qquad\qquad\quad (\mathbf{begin} \\
\qquad\qquad\quad\ (:=\ \sigma_1\ (+\ \sigma_1\ x)) \\
\qquad\qquad\quad\ (f\ (-\ x\ 1)))
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ \mathbf{if0}\ \sigma_4 \qquad\qquad\qquad\qquad ,\{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1\}\rangle \\
& \qquad \sigma_1 \\
& \qquad (\mathbf{begin} \\
& \qquad\ (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_4)) \\
& \qquad\ (\sigma_3\ (-\ \sigma_4\ 1)))
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ (\mathbf{begin} \qquad\qquad\qquad ,\{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1\}\rangle \\
& \quad (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_4)) \\
& \quad (\sigma_3\ (-\ \sigma_4\ 1)))
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ (\mathbf{begin} \qquad\qquad ,\{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1\}\rangle \\
& \quad (:=\ \sigma_1\ (+\ 0\ 1)) \\
& \quad (\sigma_3\ (-\ \sigma_4\ 1)))
\end{array}
$$

$$
\twoheadrightarrow \quad \langle\ (\sigma_3\ (-\ \sigma_4\ 1))\ ,\{\sigma_1 = 1, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1\}\rangle
$$

$$
\twoheadrightarrow \quad \langle\ (\sigma_3\ 0)\ ,\{\sigma_1 = 1, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1\}\rangle
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ \mathbf{if0}\ \sigma_6 \qquad\qquad\qquad\qquad ,\ \{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1, \\
& \qquad \sigma_1 \qquad\qquad\qquad\qquad\qquad\quad \sigma_5 = \mathbf{Y}F, \sigma_6 = 0\}\ \rangle \\
& \qquad (\mathbf{begin} \\
& \qquad\ (:=\ \sigma_1\ (+\ \sigma_1\ \sigma_6)) \\
& \qquad\ (\sigma_5\ (-\ \sigma_6\ 1)))
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ \sigma_1,\ \{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1, \\
& \qquad \sigma_5 = \mathbf{Y}F, \sigma_6 = 0\}\ \rangle
\end{array}
$$

$$
\begin{array}{ll}
\twoheadrightarrow & \langle\ 1,\ \{\sigma_1 = 0, \sigma_2 = \mathbf{Y}F, \sigma_3 = \mathbf{Y}F, \sigma_4 = 1, \\
& \qquad \sigma_5 = \mathbf{Y}F, \sigma_6 = 0\}\ \rangle
\end{array}
$$

Garbage Collection

In the preceding example, we have to allocate two new slots in the store whenever F is called: one for the f argument and one for x. But after F makes a recursive call, neither of the slots is ever used again. For example, σ_2 is never used in the second half of the evaluation. We could have predicted that σ_2 will never be used again: inspecting the expression and store half-way through the evaluation, we could see that σ_2 is not *reachable* from the expression.

In contrast, σ_1 is used all the way to the end of the evaluation. At the point where the expression to evaluate is $(\sigma_3\ 0)$, σ_1 does not appear in the expression. However, σ_3 contains the expression F, and F includes a reference to σ_1. Thus, even at the expression $(\sigma_3\ 0)$, we can see that σ_1 might be used again.

Since we might want to run the example program for an input much larger than 1, an evaluator for our language must somehow forget about useless slots. Otherwise, a program that we expect to run in constant space will keep accumulating slots on each function call.

Forgetting unused slots is the essence of *garbage collection* in languages like Java and Scheme. We can formalize the notion of garbage collection in terms of our State ISWIM language. Define $LS(\langle M, \Sigma\rangle)$, the *live slots* of $\langle M, \Sigma\rangle$, as the smallest set that obeys the following rules:

- If M contains σ, then $\sigma \in LS(\langle M, \Sigma\rangle)$.
- If $\sigma \in LS(\langle M, \Sigma\rangle)$, $\Sigma(\sigma) = V$, and V contains σ', then $\sigma' \in LS(\langle M, \Sigma\rangle)$.

Any σ that is in $\mathrm{dom}(\Sigma)$ but not in $LS(\langle M, \Sigma\rangle)$ is a garbage slot. We can add a rule to our State ISWIM semantics that allows the removal of garbage slots:

$$\begin{array}{lcl} \langle M, \Sigma[\sigma \leftarrow V]\rangle & \xrightarrow{gc} & \langle M, \Sigma\rangle \\ \quad \text{if } \sigma \notin LS(\langle M, \Sigma\rangle) & & \end{array}$$

Adding this rule to our semantics makes it non-deterministic, which means that the programmer does not know when a garbage collection will occur. While this non-determinism is not really desirable, it is realistic!

More importantly, if we ignore memory constraints, an evaluation of M with as many garbage collection steps as possible will always produce the same result as an evalution of M without any garbage collection steps. In other words, the garbage collector *proves* that certain slots will never contribute to evaluation, so that it is safe to remove them from the store.