Comp 311
Principles of Programming Languages
Lecture 16
Church and State: Supporting Assignment

Corky Cartwright

September 30, 2009

What Is Assignment?

- Assignment is rebinding (changing the value of) a variable in the current environment. This process is also called mutation since the environment is destructively changed.
- Nearly all practical programming languages include operations for mutating the values of program variables and data structures. Only plausible exception is Haskell, but is it really practical?
- To model this feature in LC, we will add an assignment operation to the language with syntax

  (set! x M)

  and the abstract representation

  (define-structure (setter lhs rhs))

  where **x** is any lambda-bound identifier.
- Assignment set! enables us to model changing events in the real world.

How Do We Define the Semantics of Assignment Using a Meta-Interpreter?

Two common approaches:

- Use mutation in the meta-language.
- Add a parameter to the eval function representing a *store* which maps *locations* to values. The *environment* maps assignable (mutable) variables to locations. What is a *location*? An element of a specified denumerable set, typically the natural numbers (akin to machine addresses).

Trade-offs: the second approach is pure but ugly. It makes interpreters look like compilers. Yuck!
Implication: assignment is inherently ugly from a semantic perspective.

Using Mutation to Define Mutation

Key intuition: implementation is easy *provided* environment sharing-relationships are modeled correctly. A nested environment shares its parent environment!

Observation: there is no straightforward way to support assignment if environments are represented as functions. Why? Assignment must update shared bindings but functions do not support any sharing relationships. Linked lists (and other concrete data structures) do!

To change the value associated with a variable **x**, we must bind a different value to the variable **x**. We can accomplish this by including a clause in MEval case-split of the form:

((setter? M) <change the environment>)

But how do we do this?

Using Mutation to Define Mutation cont.

- To make variables assignable, we need to change what they stand for.
- Variables cannot be directly associated with values; rather, they must be associated with an object which can be modified to hold a different value.
- What kind of object can we use?
  - In Scheme, a particularly apt choice is to use a box to hold the value of each variable. Then we can use mutation on Scheme boxes to change the value of the second field.
  - In Java, the value field in a Binding object simply has to be mutable.
- Moral: variables must stand for boxes (mutable cells).
- Comment: assignment languages like Java implicitly use boxes almost everywhere, but these boxes are *not* objects. They cannot be passed as values.

Revising Our Meta-Interpreter

We must revise the clause that binds new variables (which in LC are only introduced in λ-expressions):

```
((app? M)
 (Apply
   (Eval (app-rator M) env)
   (box (Eval (app-rand M) env))))  ;; box is a constructor
```

Since variables are now bound to boxes containing values, we must change the code that for evaluating variables:

```
((var? M)
 (unbox (lookup (var-name M) env))) M)
```

We are finally ready to add the clause for assignment:

```
((setter? M)
 (set-box! (lookup (setter-lhs M) env)
           (Eval (setter-rhs M) env)))
```

Can Boxes Be Values?

- Yes. Many languages support some formulation of this concept. But the details can be delicate because we must know from context whether a variable **x** means its value or the enclosing box.
- Traditional "limited" approach: support call-by-reference as a parameter passing mechanism. The formal parameter declaration includes "type" information stating that call-by-reference should be used. Examples: **var** parameters in Pascal, **ref** parameters in C++.
- Cleaner "comprehensive" approach: treat boxes as ordinary values (as in ML) or, in lower level languages, treat pointers as values (as in C). But there is a cost: these boxes/pointers must be explicitly dereferenced to get the associated values. (C complicates matters by automatically dereferencing variables in some ["right-hand"] contexts but not in others ["left-hand contexts"]).
- Conceptually, the ML convention is much simpler *but* it requires explicit dereferencing (using the unary prefix operator **!**) whenever we want the value of the variable. Our imperative (assignment) extension to Jam follows this path.