Comp 411
Principles of Programming Languages
Lecture 15
Church and State: Supporting Assignment

Corky Cartwright

February 26, 2021

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 incorporate this feature in LC, we add an assignment operation to the language with syntax (taken from Scheme/Racket)

  ```
  (set! x M)
  ```

  and the abstract representation

  ```
  (define-struct (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:

1. Use mutation in the meta-language
2. Add another parameter to the eval function representing a *store* that maps *locations* to values. The *environment* maps assignable (mutable) variables [symbols] to locations. What is a *location*? An element of a specified denumerable set, typically the natural numbers (akin to machine addresses!). Such an interpreter is called *store-passing*.

Implications:

- Trade-offs: the second approach is pure but ugly. It makes interpreters look like compilers; identifiers stand for addresses.
- Conclusion: assignment is inherently ugly from a semantic perspective. Store-passing is used in denotational semantics for imperative (mutation supporting) languages but it gives no insight on how to build an efficient implementation. Meta-interpreters that rely on mutation are much closer to efficient language implementations because all modern computers support mutation (updating the value at an address in memory!)

Two Different Formulations of Assignment

Assignable Variables

- Mutate (change) bindings in environment.
- The semantics of assignment critically depend on the fact that extensions of an environment share that environment.
- Racket Scheme/C++/Java/Scala/Swift use this formulation.

Mutable Cells

- Cells (boxes) are *values*. In essence, the data domain is augmented by a new unary constructor called `box` or `ref`. The *only* mutable data cells are these cells.
- ML/Haskell uses this formulation.

Comparing the Two Different Formulations

Advantage of Assignable Variables

- Simpler notation but the *cells/boxes* holding the values of variables typically are not data objects which forces extra machinery (typically a prefix operator like `@` in C) in parameter passing (such as call-by-reference) or a distinction between *left-hand* and *right-hand* evaluation.
- If cells are data objects (*e.g.*, pointers as in C) the internals of the language implementation are exposed (as in C).

Advantage of Mutable Cells

- Mutable cells are simply a special form of data.
- The design of the language is unaffected otherwise.
- Simulating *call-by-reference* is trivial; *call-by-name* requires a bit of extra work, but not much. *Call-by-need* is more work but it is generally not supported in languages with assignable variables either.

Using Mutation to Define Mutation

Key intuition: implementation is easy, *provided* that environment sharing-relationships are modeled correctly. A nested environment shares its parent environment representation!
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 directly support sharing relationships. Linked lists (and other concrete data structures) foster sharing!

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 the `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 support changing the values 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* (a built-in struct with a single field to hold the value of each variable. Then we can use mutation on Scheme boxes to change the value of the field. Note that we can also use closures and mutate local variables in these closures, which is an important trick featured in *SICP* (*Structure and Interpretation of Progams*).
- In Java, the value field in a **`Binding`** object simply has to be mutable.
- Moral: variables should 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. They are not visible data values except via the Java reflection facility which is ugly and often breaks portability and backward compatibility.

Revising Our Meta-Interpreter

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

```
((app? M)
 (Mapply
   (MEval (app-fp M) env)
      (box (MEval (app-ap 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)
                       (MEval (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 either its value or the enclosing box. In ML, it is trivial.
- Traditional "limited" approach: support call-by-reference as a parameter passing mechanism. The formal parameter declaration includes an attribute stating that call-by-reference should be used. Examples: `var` parameters in Pascal, reference (tagged with `&`) parameters in C++. Variables passed by reference are interpreted differently (as the box or its contents) depending on context.
- Cleaner "comprehensive" approach: treat boxes as ordinary values (as in ML) or, in lower level languages, pointers as values (as in C). But there is a cost: these boxes/pointers must be explicitly dereferenced to get the associated values. In C, the data model is ultimately machine memory and explicit use of pointers is perilous; tiny mistakes can cause catastrophic behavior (screen of blue death [SOBD]) All "algol-like" languages (including C/C++) depend on program context to determine when variables are automatically dereferenced. Conceptually, the ML convention is much simpler *but* it requires explicit dereferencing (using the unary prefix operator `!`) whenever we want the value of the mutable variable.