Comp 411
Principles of Programming Languages
Lecture 14
Church and State: Supporting Assignment

Corky Cartwright

October 3, 2012

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 abstract syntax

```
case class Assign(left: Symbol, right: Exp) extends Exp
```

where **left** is the name of a lambda-bound variable and **right** is an expression specifying the new value of the variable.

- Assignment enables us to model changing events in the real world.

How Do We Define the Semantics of Assignment?

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).

Implications:

- 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.

We will ignore the *store-passing* approach and use mutation in the metalanguage.

Adding Assignable Variables Without Boxes

Key intuition: we can easily revise our call-by-value LC interpreter to support assignment by mutating the bindings in the environment *provided* environment sharing-relationships are modeled correctly. There are two ways to manage sharing. One is to use an environment representation where an extended environment shares its parent environment! This property does not hold in general for Scala mutable **`Map`** types (or function types)! But it does hold for the mutable **`ListMap`** type. The other approach is to add a level of indirection in the representation of bound values in the environment. We can explicitly create boxes holding values and bind identifiers to boxes (containing values) instead of values. The disadvantage of this approach is inefficiency: every variable reference incurs an extra memory reference.

Let's explore the environment sharing approach first. 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 **`eval`** pattern-match of the form (and ensuring that environments are represented by **`ListMap`**):

```
case Assign(left, right) =>
  val value = eval(right, env)
  env.put(left, value)  (List[Pair(Symbol,Value)])
  Undefined
```

We also must add a case class definition for **`Undefined`** as a form of **`Value`** to provide an appropriate return value for assignments (**`put`** method calls)

Adding Assignable Variables Using Boxes

- To make variables assignable using an immutable environment representation like functions, we need to change 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 type) 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 a `BoundVal` 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 in the program. They cannot be passed as values.
- In our Scala code stipulating a `mutable.ListMap` representation for environments, we implicitly relied on the fact that the representation includes a mutable cell for the value of each key.

Adding Assignable Variables Using Boxes cont.

To support boxing, we need to introduce **`Box`** as an implementation type. These boxes are not values.

- Moral: variables must stand for boxes (mutable cells).
- Comment: assignment languages like Java implicitly use boxes almost everywhere, but these boxes are *not* objects in the program. They cannot be passed as values. The same is true in our imperative LC language

We revise our LC interpreter by adding the following data definitions:

```
case object Undefined extends Value
case class Box(var contents: Value) extends Value
case class Assign(left: Symbol, right: Exp) extends Exp
```

To define the semantics of assignment statements, we add the following pattern matching clause to **`eval`**:

```
case Assign(left, right) =>
     val box = env(left)
     box.contents = eval(right, env)
     Undefined
```

Finally, we change the semantics variable references to use boxes by revising the case clause for variables:

```
case Var(s) => env(s).contents
```

And the semantics of **`Lambda`** binding by revising the definition of **`bind`**:

```
def bind(s: Symbol, v: Value, env: Env): Env = env + (s -> Box(v))
```

Mutable Cells Instead of Assignable Variables

The language consisting of functional LC plus assignment is not very attractive no matter which implementation we choose: (i) shared environments with destructive environment updating or (ii) immutable environments with boxed values. Why? Because boxes (or mutable environment fields) are present by we cannot access them as values. In our next lecture, we will explore the common language extension to support limited access to the boxes holding values. But there is another way to support mutation (imperativity) that elegantly avoids the hidden box problem. We can add mutable boxes explicitly to our LC language. Only the contents of such boxes are mutable. This approach, pioneered in ML, is easy to implement but requires more abstract syntax because we must add operations to create boxes, extract their contents, and update their contents.

Assignment Using Explicit Boxes

To add mutation via explicit boxes to LC, we must define boxes as value constructors (`Box`) and add AST classes for creating boxes (`MakeBox`), updating boxes (`Put`), and getting the contents of boxes (`Unbox`)

```
case class Box(var contents: Value) extends Value

case class MakeBox(exp: Exp) extends Exp
case class Put(target: Exp, exp: Exp) extends Exp
case class Unbox(exp:Exp) extends Exp
```

Similarly, we must add clauses to our interpreter to handle the three new AST classes:nce variables are now bound to boxes containing values, we must change the code that for evaluating variables:

```
case MakeBox(exp) => Box(eval(exp, env))
case Put(target, exp) =>
  val box = eval(target, env)
  if (! box.isInstanceOf[Box]) throw new
    IllegalArgumentException("Attempt to update non-box " + box)
  box.asInstanceOf[Box].contents = eval(target, env)
  Undefined
case Unbox(exp) =>
  val result = eval(exp, env)
  if (! result.isInstanceOf[Box]) throw new
    IllegalArgumentException("Attempt to unbox the non-box " + result)
  result.asInstanceOf[Box].contents
```

Boxes As Values

- In our second version of imperative LC (with explicit mutable boxes), boxes are data values and only the contents of a box may be mutated. Many languages support some mechanism for accessing the boxes holding values. These language designs fall into two different categories.
  - Traditional languages with reference parameters. Box values are only created by passing an argument to a method/subroutine where the corresponding formal parameter is declared of reference type. This approach is delicate because context determines whether a variable name (more generally any expression that can appear on the left-hand side of an assignment) denotes a box or the contents of the box Example var parameters in Pascal.
  - Clean "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. C and other "algol-like" languages complicates matters automatically dereference 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. So does our second version of imperative LC since it is patterned after ML. In addition, note that variable bindings are immutable. Only the contents of boxes can change. Hence, a "mutable" variable is a variable that is immutably bound to a box.

Assignable Variables vs. Mutable Boxes

- Which approach to extending LC to support assignment is more general (in terms of what methods we can define). Consider our example.
- Which was simpler to code? Ordinary assignable variables are simple from an implementation perspective, but peculiar from the perspective of the programming language because the boxes for variables are not available as values in programs. Many languages with assignable variables support call-by-reference, which provides llimited access to the boxes corresponding to variables, for this reason.
- The primary disadvantage of explicit mutable boxes is that the programmer must specify unboxing explicitly or the language must perform coercions, increasing the complexity and unpredictability of program behavior. IMO, Scala should have incorporated mutable boxes instead of assignable variables; programmers annoyed by explicit boxing and unboxing could use the `implicit` mechanism to coerce `Box[T]` to `T`. Scala adopted assignable (`var`) variables instead, presumably to maintain consistency with Java.