

# **Mutable State**

February 27, 2020

Riccardo Pucella

# What we've done till now

Object languages:

- simple calculator language
- added first-class recursive functions (FUNC)
  - That language is Turing-complete
  - It can implement every possible computable function

*Aside: I showed last time that we need only single-argument anonymous functions, function calls, and conditionals. This is essentially the **lambda calculus**.*

# What's next?

Mutable state (and in general, side effects)

We add variables whose value can be changed during a computation

- Every function that refers to that variable will see the change in value

```
def show ():  
    print x + 1
```

```
x = 10  
show()  
x = x * 2  
show()
```

# Two approaches to mutable state

**ML style:** add a special class of values representing mutable variables

**C style:** every identifier is a mutable variable

*C-style mutable state can be syntactically translated to ML-style mutable state*

# **ML-style mutable state**

New kind of value: reference cell

A reference cell is a box that contains a value

You can pass the reference cell around

You can get the reference cell's content

You can put content in the reference cell

# ML-style mutable state: API

(ref 10)      create a reference cell  
                 with initial content 10

(get r)        get content of reference cell r

(put r 20)    put 20 into reference cell r

# ML-style mutable state: API

`(ref 10)`      create a reference cell  
with initial content 10

`(get r)`      get content of reference cell `r`

`(put r 20)`    put 20 into reference cell `r`

You do not call this to get a value

You call this to perform an action!

# New value: VRefCell

```
class VRefCell (val init : Value) extends Value {  
  
  var content = init  
  
  override def toString () : String =  
    "ref(" + content + ")"  
  override def isRefCell () : Boolean = true  
  
  override def getRefContent () : Value = content  
  override def putRefContent (v:Value) : Unit = {  
    content = v  
  }  
}
```



# Primop to create a reference cell

Bound to identifier ref

```
def operRefCell (vs:List[Value]) : Value = {  
    checkArgsLength(vs,1,1)  
    val init = vs(0)  
    return new VRefCell(init)  
}
```

# Primop to get from a reference cell

Bound to identifier get

```
def operGetRefCell (vs:List[Value]) : Value = {  
    checkArgsLength(vs,1,1)  
    val r = vs(0)  
    r.checkRefCell()  
    return r.getRefContent()  
}
```

# Primop to put into a reference cell

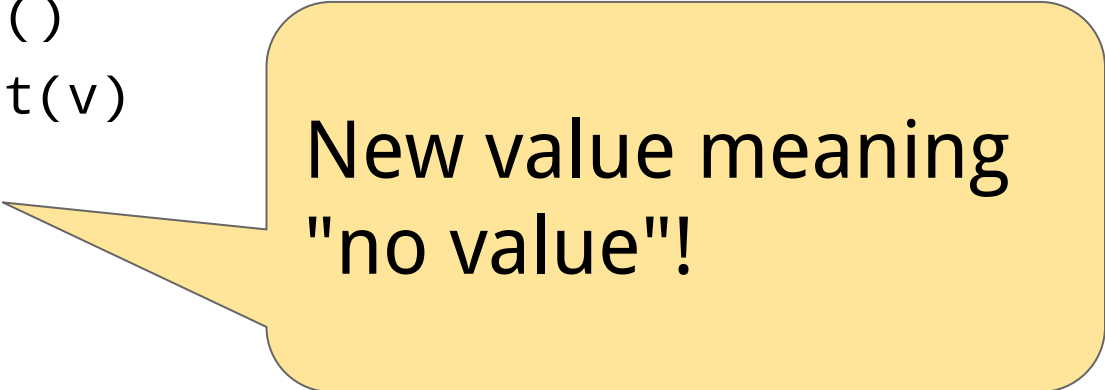
Bound to identifier put

```
def operPutRefCell (vs:List[Value]) : Value = {  
    checkArgsLength(vs,2,2)  
    val r = vs(0)  
    val v = vs(1)  
    r.checkRefCell()  
    r.putRefContent(v)  
    return VNone  
}
```

# Primop to put into a reference cell

Bound to identifier put

```
def operPutRefCell (vs:List[Value]) : Value = {  
    checkArgsLength(vs,2,2)  
    val r = vs(0)  
    val v = vs(1)  
    r.checkRefCell()  
    r.putRefContent(v)  
    return VNone  
}
```



New value meaning  
"no value"!

# Sequencing

Performing an action does not return a value

In order to mix action-performing expressions (sometimes known as *statements*) and other expressions, it helps to have a way perform a sequence of actions *before* returning a value

```
(do e1 e2 e3 ... e)
```

# EDo

```
class EDo (val es : List[Exp]) extends Exp {  
  ...  
  
  def eval (env : Env) : Value = {  
    var v : Value = VNone  
    for (e <- es) {  
      v = e.eval(env)  
    }  
    return v  
  }  
}
```

# Other primitive operations

While we're at it, we can introduce other operations that perform *actions* like `operSetRefCell`, such as printing values  
Bound to identifier `print`

```
def operPrint (vs:List[Value]) : Value = {  
    for (v <- vs) {  
        print(v)  
        print(" ")  
    }  
    println(" ")  
    return VNone  
}
```

# Example

```
(let ((result (ref 0)))  
  (do (print (get result))  
      (put result 10)  
      (print (get result))  
      (put result (+ (get result) 30))  
      (get result)))
```



# Actions cause side effects

And side effects are observable

Order of evaluation is now relevant!

```
(let ((x (ref 0)))  
  (+ (get x) (do (put x 10)  
                  (get x)))))
```

This evaluates to 10 left-to-right,  
but 20 right to left!

# Iteration

Once you have mutable state, other control structures for performing actions become reasonable.

A while loop repeats a sequence of actions until a particular condition is true.

(What is the equivalent in a functional world?)

# EWhile

```
class EWhile (val cond: Exp, val body: Exp) extends Exp {  
  ...  
  
  def eval (env : Env) : Value = {  
    var vc = cond.eval(env)  
    vc.checkBoolean()  
    while (vc.getBool()) {  
      val v = body.eval(env)      // result never used  
      vc = cond.eval(env)  
      vc.checkBoolean()  
    }  
    return VNone  
  }  
}
```

# Example

```
(let ((result (ref 0))
      (count (ref 10)))
  (do (while (not (= 0 (get count)))
      (do (put result (+ (get result)
                        (get count)))
          (put count (- (get count) 1))))
    (get result)))
```

# EDo as a syntactic transformation

*Assuming eager evaluation*

```
(do e1 e2 e3 ... e)
```

→

```
(let ((unused e1))  
  (let ((unused e2))  
    (let ((unused e3))  
      ...  
      e)))
```

# EWhile as a syntactic transformation

`(while cond exp) =`

`→`

```
(let ((while (fun w ()  
                (if cond (do exp  
                           (w))  
                        none))))  
      (while)))
```

# Analysis

- It is entirely straightforward to implement mutable state via reference cells
- But it's heavy
  - You need to identify the things you want to mutate
  - You need to explicitly "get" a cell to get to its value
  - Expressions using the content of reference cells become hard to read
- Best approach when most of your code is functional and you need just a little bit of state (e.g., React)





# C-style mutable state for us

Our previous example with all bindings mutable:

```
(let ((result 0)
      (count 10))
  (do (while (not (= count 0))
      (do (set result
              (+ result count))
          (set count (- count 1))))
    result))
```

# Option 1: Modify environments

Requires changing how we evaluate the abstract representation

- Change implementation of environments so that we can update the value of a binding

Pretty straightforward

Some subtleties: we have to make sure we don't copy environments and lose sharing

# Option 2: Syntactic transformation

No change to the evaluation mechanism

Transform code into an abstract representation with explicit reference cells

Intuition:

- every binding uses a reference cell
- $x$  in an expression is basically `(get x)`
- `(set x E)` is basically `(put x E)`

# Transformation function

Transformation of a surface syntax expression into an expression using explicit reference cells:

$$\llbracket expr \rrbracket$$
$$\llbracket integer \rrbracket = integer$$
$$\llbracket boolean \rrbracket = boolean$$
$$\llbracket (if\ expr1\ expr2\ expr3) \rrbracket = (if\ \llbracket expr1 \rrbracket\ \llbracket expr2 \rrbracket\ \llbracket expr3 \rrbracket)$$
$$\llbracket (expr1\ expr2\ \dots) \rrbracket = (\llbracket expr1 \rrbracket\ \llbracket expr2 \rrbracket\ \dots)$$
$$\llbracket (do\ expr1\ \dots\ exprk) \rrbracket = (do\ \llbracket expr1 \rrbracket\ \dots\ \llbracket exprk \rrbracket)$$
$$\llbracket (while\ expr1\ expr2) \rrbracket = (while\ \llbracket expr1 \rrbracket\ \llbracket expr2 \rrbracket)$$

# Transformation function

## Interesting cases:

```
[[ identifier ]] = (get identifier)
```

$$\llbracket (\text{fun } (s1 \dots) \text{ expr}) \rrbracket = (\text{fun } (s1 \dots) \\ \quad (\text{let } ((s1 \text{ (ref } s1)) \dots) \\ \quad \quad \llbracket \text{expr} \rrbracket))$$
$$\llbracket (\text{let } ((s1 \text{ } expr1) \dots) \text{ } expr) \rrbracket = (\text{let } ((s1 \text{ } (\text{ref } \llbracket expr1 \rrbracket)) \dots) \llbracket expr \rrbracket)$$
$$\llbracket (\text{set } identifier \text{ expr}) \rrbracket = (\text{put } identifier \llbracket expr \rrbracket)$$