# Mutable State

October 11, 2016

Riccardo Pucella

# What we’ve done till now

Object languages:

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

*Aside: we’ve shown in Homework 5 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 translated to ML-style mutable state*

# ML-style mutable state

New kind of value: reference cell

A reference cell is like a box that contains a value.

You can pass the reference cell around.
You can access the reference cell's content
You can change the reference cell's content

# ML-style mutable state: API

| | |
|---|---|
| `(ref 10)` | create a reference cell with initial content 10 |
| `(deref r)` | gets content of reference cell `r` |
| `(update! r 20)` | update content of reference cell `r` to 20 |

# ML-style mutable state: API

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

`(deref r)` gets content of reference cell `r`

`(update! r 20)` update content of reference cell `r` to 20

You do not call this to get a value

You call this to perform an action!

# VRefCell (and VNone)

```
class VRefCell (Value):

    def __init__ (self,initial):
        self.content = initial
        self.type = "ref"



class VNone (Value):
    # used to represent No Value
    def __init__ (self):
        self.type = "none"
```

# ERefCell

```
class ERefCell (Exp):

    def __init__ (self,initialExp):
        self._initial = initialExp

    def eval (self,env):
        v = self._initial.eval(env)
        return VRefCell(v)
```

# Primitive operations on VRefCell

```
def oper_deref (v1):
    if v1.type == "ref":
        return v1.content
    raise Exception ("Runtime error: not a ref cell")


def oper_update (v1,v2):
    if v1.type == "ref":
        v1.content = v2
        return VNone()
    raise Exception ("Runtime error: not a ref cell")
```

# Other primitive operations

While we're at it, we can introduce other operations that perform *actions*, like `oper_update`:

```
def oper_print (v1):
    print v1
    return VNone()
```

# Sequencing

Performing an action does not return a value

n order to mix action-performing expressions 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 (Exp):

    def __init__ (self,exps):
        self._exps = exps

    def eval (self,env):
        v = VNone()
        # evaluate exps, return last value
        for e in self._exps:
            v = e.eval(env)
        return v
```

# Example

```
(let ((result (ref 0)))
  (do (print! (deref result))
      (update! result 10)
      (print! (deref result))
      (update! result (+ (deref result) 30))
      (deref result)))
```

# Actions cause side effects

And side effects are observable

Order of evaluation is now relevant!

```
(let ((x (ref 0)))
  (+ (deref x) (do (update! x 10)
                   (deref 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 (Exp):
    def __init__ (self,cond,exp):
        self._cond = cond
        self._exp = exp

    def eval (self,env):
        c = self._cond.eval(env)
        if c.type != "boolean":
            raise Exception ("Runtime error: while")
        while c.value:
            self._exp.eval(env)
            c = self._cond.eval(env)
            if c.type != "boolean":
                raise Exception ("Runtime error: while")
        return VNone()
```

# Example

```
(let ((result (ref 0)))
     ((count (ref 10)))
  (do (while (not (zero? (deref count)))
             (do (update! result (+ (deref result)
                                     (deref count)))
                 (update! count (- (deref count)
                                    1))))
      (deref result)))
```

# EDo is not necessary

*Assuming eager evaluation*

```
(do e1 e2 e3 … e)
```

$\longrightarrow$

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

# EWhile is not necessary

```
(while cond exp) =

    →

        (let ((while (function 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 dereference a cell to get to its value
  - Expressions using the content of reference cells get hard to read

- Best to use when most of your code is functional and you need just a little bit of state (e.g., React)

# C-style mutable state

By default, every binding is mutable

```
var x = 10;              ← declaration with initialization

while (x != 0) {         ← get the value in x
  print x;               ← get the value in x
  x <- x - 1;            ← get the value in x
                           and store result in x
}
```

# Statements versus expressions

Expressions evaluate down to values:

```
- x + 1
- x != 0
```

Statements perform actions:

```
- x = x + 1;
- while ...
- print x;
```

Declarations set up bindings:

```
- var x = 10;
```

# C-like surface syntax

Expressions are pretty much as before:

```
expr ::= integer
         true
         false
         identifier
         ( if expr expr expr )
         ( function ( name ... ) expr )
         ( expr expr ... )
```

# C-like surface syntax

Added declarations and statements:

```
decl ::= var name = expr ;

stmt ::= if expr stmt else stmt
         while expr stmt
         name <- expr ;
         print expr ;
         { decl ... stmt ... }
```

(We could use a more natural syntax for expressions, as in Homework 3)

## Exercise: design an abstract representation

- Value nodes (as before)

- Expr nodes (as before)
  - have `eval()` method that returns a value

- Stmt nodes
  - have `execute()` method that does not return a value

This is all pretty straightforward

# Alternative: front-end transformations

- Use a C-like surface syntax
- But transform it into the FUNC abstract representation with reference cells

Intuition:

- every binding uses a reference cell
- `{ var x = E; … S1; S2; ... }` is basically
  `(let ((x (ref E)) …) (do S1 S2 …))`
- `x` in an expression is basically (deref x)
- `x <- E ;` is basically `(update! x E)`

# Transformation functions

Transformation of a surface syntax expression into abstract representation:

$\mathrm{E}[\![ \mathit{expr} ]\!]$

Transformation of a surface syntax statement into abstract representation:

$\mathrm{S}[\![ \mathit{stmt} ]\!]$

# Expressions transformation

E⟦*integer*⟧ = `EValue(VInteger(`*integer*`))`

E⟦*boolean*⟧ = `EValue(VBoolean(`*boolean*`))`

E⟦*identifier*⟧ = `EPrimCall(oper_deref,[EId(`*identifier*`)])`

E⟦`(if` *expr1 expr2 expr3*`)`⟧ = `EIf(`E⟦*expr1*⟧,E⟦*expr2*⟧,E⟦*expr3*⟧`)`

This is pretty much the way we dealt with expressions in FUNC, except for identifiers

The other minor difference is functions...

# Expressions transformation

```
E⟦(function (name1 …) expr)⟧ =
     EFunction([name1,...],
               ELet([(name1,ERefCell(EId(name1))),...],
                    E⟦expr⟧))

E⟦(expr1 expr2 …)⟧ = ECall(E⟦expr1⟧,[E⟦expr2⟧,...])
```

# Statements transformation

S⟦ if *expr* *stmt1* else *stmt2* ⟧ =
   EIf(E⟦*expr*⟧,S⟦*stmt1*⟧,S⟦*stmt2*⟧)

S⟦ while *expr* *stmt* ⟧ =
   EWhile(E⟦*expr*⟧,S⟦*stmt*⟧)

S⟦ *name* <- *expr* ⟧ =
  EPrimCall(oper_update,[EId(*name*),E⟦*expr*⟧])

S⟦ print *expr* ⟧ =
  EPrimCall(oper_print,E⟦*expr*⟧)

S⟦ { var *name1* = *expr1*; …; *stmt1*; … } ⟧ =
  ELet([(*name1*,ERefCell(E⟦*expr1*⟧)),...],
      EDo([S⟦*stmt1*⟧, …))

# Example

```
{
  var count = 10;
  var result = 0;
  while (not (zero? count)) {
    result <- (+ result count);
    count <- (- count 1);
  }
  print count;
}
```

# Example

```
ELet([(count,ERefCell(EValue(VInteger(10)))),
      (result,ERefCell(EValue(VInteger(0))))],
     EDo([EWhile(ECall(EPrimCall(oper_deref,[EId("not")]),
                  [ECall(EPrimCall(oper_deref,[EId("zero?")]),
                         [EPrimCall(oper_deref,[EId("count")])])])]),
             EDo([EPrimCall(oper_update,
                           [EId("result"),
                            ECall(EPrimCall(oper_deref,[EId("+")]),
                                  [EPrimCall(oper_deref,[EId("result")]),
                                   EPrimCall(oper_deref,[EId("count")])])
                           ]),
                  EPrimCall(oper_update,
                           [EId("count"),
                            ECall(EPrimCall(oper_deref,[EId("-")]),
                                  [EPrimCall(oper_deref,[EId("count"]),
                                   EValue(VInteger(1))])])])])),
          EPrimCall(oper_print,[EPrimCall(oper_deref,[EId("count")])])]))
```