CS4450/7450
AoPL, Chapter 6: Computational Strategies
Principles of Programming Languages

Dr. William Harrison

University of Missouri

November 18, 2016

Announcements

- We're continuing with William Cook's online textbook, *Anatomy of Programming Languages*. It is available here. We're in Chapter 6.

Outline for section 1

Outline for section 2

Mutable State

Mutable State

Mutable state means that the state of a program changes or mutates: that a variable can be assigned a new value or a part of a data structure can be modified.

Ex: The values of `x` and `i` change over time:

```
x = 1;
for (i = 2; i <= 5; i = i + 1) {
  x = x * i;
}
```

Addresses

Addresses

An address identifies a mutable container that stores a single value, but whose contents can change over time. Addresses are sometimes called locations.

Addresses

Addresses

An address identifies a mutable container that stores a single value, but whose contents can change over time. Addresses are sometimes called locations.

Operation	Meaning
`mutable e`	Creates a mutable cell with initial value given by `e`
`@e`	Accesses contents stored at address `e`
`a = e`	Updates contents at address `a` to be value of expression `e`

Example

```
x = mutable 1;
for (i = mutable 2; @i <= 5; i = @i + 1) {
  x = @x * @i;
}
```

New Values and Expressions

```haskell
data Value = IntV  Int
           | BoolV Bool
           | ClosureV String Exp Env
           | AddressV Int          -- new
  deriving (Eq, Show)

type Memory = [Value]
```

New Values and Expressions

```
data Value = IntV  Int
           | BoolV Bool
           | ClosureV String Exp Env
           | AddressV Int          -- new
  deriving (Eq, Show)

type Memory = [Value]

data Exp = ...
         | Mutable   Exp          -- mutable e
         | Access    Exp          -- @a
         | Assign    Exp Exp      -- a = e
```

Illustrating Memory Operations

Step	Memory
start	[]

Illustrating Memory Operations

Step	Memory
start	[]
`x = mutable 1;`	[1]

Illustrating Memory Operations

Step	Memory
start	[]
`x = mutable 1;`	[1]
`i = mutable 2;`	[1,2]

Illustrating Memory Operations

Step	Memory
start	[]
`x = mutable 1;`	[1]
`i = mutable 2;`	[1,2]
`x = @x * @i;`	[2,2]

Illustrating Memory Operations

Step	Memory
start	[]
`x = mutable 1;`	[1]
`i = mutable 2;`	[1,2]
`x = @x * @i;`	[2,2]
`i = @i + 1;`	[2,3]

Illustrating Memory Operations

Step	Memory
start	[]
`x = mutable 1;`	[1]
`i = mutable 2;`	[1,2]
`x = @x * @i;`	[2,2]
`i = @i + 1;`	[2,3]
`x = @x * @i;`	[6,3]
`i = @i + 1;`	[6,4]
`x = @x * @i;`	[24,4]
`i = @i + 1;`	[24,5]
`x = @x * @i;`	[120,5]
`i = @i + 1;`	[120,6]

Operations on Memory

Accessing Memory:

```
access i mem = mem !! i
```

Operations on Memory

Accessing Memory:

```
access i mem = mem !! i
```

Updating Memory:

```
update :: Int -> Value -> Memory -> Memory
update addr val mem =
  let (before, _ : after) = splitAt addr mem in
    before ++ [val] ++ after
```

Operations on Memory

Accessing Memory:

```
access i mem = mem !! i
```

Updating Memory:

```
update :: Int -> Value -> Memory -> Memory
update addr val mem =
  let (before, _ : after) = splitAt addr mem in
    before ++ [val] ++ after
```

```
ghci> :t splitAt
   splitAt :: Int -> [a] -> ([a], [a])
ghci> splitAt 3 "abcdefg"
   ("abc","defg")
ghci> splitAt 0 "abcdefg"
   ("","abcdefg")
```

Whither `evaluate`?

- Currently:

```
evaluate :: Exp -> Env -> Value
```

Whither `evaluate`?

- Currently:

```
evaluate :: Exp -> Env -> Value
```

- Defn. A **stateful computation** is a function that takes an input state and returns a value and an output state.

Whither `evaluate`?

- Currently:

```
evaluate :: Exp -> Env -> Value
```

- Defn. A **stateful computation** is a function that takes an input state and returns a value and an output state.
- Now a stateful computation:

```
evaluate :: Exp -> Env -> Memory -> (Value,Memory)
```

Whither `evaluate`?

- Currently:

```
evaluate :: Exp -> Env -> Value
```

- Defn. A **stateful computation** is a function that takes an input state and returns a value and an output state.
- Now a stateful computation:

```
evaluate :: Exp -> Env -> Memory -> (Value,Memory)
```

- Generalizing:

```
type Stateful a = Memory -> (a,Memory)
evaluate :: Exp -> Env -> Memory -> Stateful Value
```

Evaluation Rules

```
evaluate (Mutable e) env mem =
   let
      (ev, mem') = evaluate e env mem
   in
      (AddressV (length mem'), mem' ++ [ev])
```

Evaluation Rules

```
evaluate (Mutable e) env mem =
   let
      (ev, mem') = evaluate e env mem
   in
      (AddressV (length mem'), mem' ++ [ev])

evaluate (Access a) env mem =
  let
     (AddressV i, mem') = evaluate a env mem
  in
     (access i mem', mem')
```

Evaluation Rules

```
evaluate (Mutable e) env mem =
   let
      (ev, mem') = evaluate e env mem
   in
      (AddressV (length mem'), mem' ++ [ev])

evaluate (Access a) env mem =
  let
     (AddressV i, mem') = evaluate a env mem
  in
     (access i mem', mem')
```

Evaluation Rules (cont'd)

```
evaluate (Assign a e) env mem =
  let
     (AddressV i, mem') = evaluate a env mem
  in
     let
        (ev, mem'') = evaluate e env mem'
     in
        (ev, update i ev mem'')

evaluate (Binary op a b) env mem =
  let
     (av, mem') = evaluate a env mem
  in
     let
        (bv, mem'') = evaluate b env mem'
     in
        (binary op av bv, mem'')
```

A pattern is forming...

- Recall:

```
evaluate (Mutable e) env mem =
   let (ev, mem') = evaluate e env mem
   in  (AddressV (length mem'), mem' ++ [ev])
```

A pattern is forming...

- Recall:

```
evaluate (Mutable e) env mem =
   let (ev, mem') = evaluate e env mem
   in  (AddressV (length mem'), mem' ++ [ev])
```

- The pattern:

```
 \ mem ->
   let
      (val, mem') = first-part mem
   in
      next-part val mem'
```

A pattern is forming...

- Recall:

```
evaluate (Mutable e) env mem =
   let (ev, mem') = evaluate e env mem
   in  (AddressV (length mem'), mem' ++ [ev])
```

- The pattern:

```
 \ mem ->
   let
      (val, mem') = first-part mem
   in
      next-part val mem'
```

- Generalizing as a higher-order function:

```
(>>=) :: Stateful a -> (a -> Stateful b) -> Stateful b
first-part >>= next-part =
 \ mem ->
   let
      (val, mem') = first-part mem
   in
      next-part val mem'
```