Programs Languages and Techniques (CIS120)

Lecture 14
February 11, 2013

Imperative Queues
Announcements

• Homework 4 due tonight at midnight

• Midterm 1 will be in class on Friday, February 15th
  – ROOMS:
    • Towne 100 (here) last names: A – K
    • Cohen G17 last names: L – Z
  – TIME: 11:00a.m. sharp, 50 mins
  – Covers up to Feb 6th
    • no Abstract Stack Machine
Mutable Records and the ASM
Abstract Stack Machine

• Three “spaces”
  – workspace
    • contains the expression the computer is currently working with
    • Machine operation simplifies expression to value
  – stack
    • temporary storage for let bindings, function parameters and stored workspaces (function call)
    • maps variable names to values (primitive values or references to heap locations)
  – heap
    • storage area for large data structures (datatypes, tuples, first-class functions, records)
We had to go through all this abstract stack stuff to make the model of heap locations and sharing explicit.

Now we can say what it means to mutate a heap value in place.

```plaintext
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int = p2.x <- 17; p1.x
```

We draw a record in the heap like this:

- The doubled outlines indicate that those cells are mutable
- Everything else is immutable
- (field names don’t actually take up space)
allocate a record

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
  p2.x <- 17; p1.x
Allocate a Record

```
let p1 : point =  
let p2 : point = p1 
let ans : int = 
  p2.x <- 17; p1.x
```

Workspace

Stack

Heap

<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>
Let Expression

```
let p1 : point = ...;
let p2 : point = p1
let ans : int =
  p2.x <- 17; p1.x
```
Push $p_1$

```
let $p_2$ : point = $p_1$
let ans : int =
  $p_2$.x <- 17; $p_1$.x
```
let p2 : point = p1
let ans : int =
    p2.x <- 17;
    p1.x
let p2 : point =
let ans : int =
  p2.x <- 17; p1.x
let p2 : point = .
let ans : int =
p2.x <- 17; p1.x
let ans : int =
p2.x <- 17; p1.x

Note: p1 and p2 are references to the same heap record. They are *aliases* – two different names for the *same* thing.
let ans : int = p2.x <- 17; p1.x
let ans : int =
  .x <- 17; p1.x
let ans : int = 
  x <- 17; p1.x
let ans : int = (); p1.x

Assign to x field

Workspace

Stack

p1

p2

Heap

x 17

y 1
Sequence ‘;’ Discards Unit

let ans : int = ( ); p1.x

Workspace

Stack

p1

p2

Heap

x 17

y 1

CIS120/ Spring 2013
let ans : int = p1.x
let ans : int = x
let ans : int = __.x
Project the ‘x’ field

Workspace

```latex
let ans : int =
17
```

Stack

- p1
- p2

Heap

- x: 17
- y: 1

CIS120/ Spring 2013
Let Expression

Workspace

let ans : int = 17

Stack
p1
p2

Heap
x 17
y 1
Push ans

Workspace

Stack

Heap

p1

x 17

p2

y 1

ans 17

DONE!
Reference and Equality

= vs. ==
• Suppose we have two counters. How do we know whether they share the same internal state?
  – `type counter = { mutable count : int }`
  – We could increment one and see whether the other’s value changes.
  – But we could also just test whether the references alias directly.

• Ocaml uses ‘==’ to mean reference equality:
  – two reference values are ‘==’ if they point to the same data in the heap:
    `r2 == r3`
  – `not (r1 == r2)`
  – `r1 = r2`
Structural vs. Reference Equality

• Structural (in)equality: \( v_1 = v_2 \) \( v_1 \neq v_2 \)
  - recursively traverses over the structure of the data, comparing the two values’ components for structural equality
  - function values are never structurally equivalent to anything
  - structural equality can go into an infinite loop (on cyclic structures)
  - use this for comparing immutable datatypes

• Reference equality: \( v_1 == v_2 \) \( v_1 != v_2 \)
  - Only looks at where the two references point into the heap
  - function values are only equal to themselves
  - equates strictly fewer things than structural equality
  - use this for comparing mutable datatypes
Putting State to Work

Queues
A design problem

Suppose you are implementing a website to sell tickets to a very popular music event. To be fair, you would like to allow people to select seats first come, first served. How would you do it?

• Understand the problem
  – Some people may visit the website to buy tickets while others are still selecting their seats
  – Need to remember the order in which people purchase tickets

• Define the interface
  – Need a datastructure to store ticket purchasers
  – Need to add purchasers to the end of the line
  – Need to allow purchasers at the beginning of the line to select seats
  – Needs to be mutable so the state can be shared across web sessions
(Mutable) Queue Interface

module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool

  (* Add a value to the tail of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the head value and return it (if any) *)
  val deq : 'a queue -> 'a

end
Define test cases

/* Some test cases for the queue */
let test () : bool =
    let q : int queue = create () in
    enq 1 q;
    enq 2 q;
    1 = deq q
;; run_test "queue test 1" test

let test () : bool =
    let q : int queue = create () in
    enq 1 q;
    enq 2 q;
    let _ = deq q in
    2 = deq q
;; run_test "queue test 2" test
Implement the behavior

module ListQ : QUEUE = struct

  type 'a queue = { mutable contents : 'a list }

  let create () : 'a queue =
  { contents = [] }

  let is_empty (q:'a queue) : bool =
  q.contents = []

  let enq (x:'a) (q:'a queue) : unit =
  q.contents <- (q.contents @ [x])

  let deq (q:'a queue) : 'a =
  begin match q.contents with
  | [] -> failwith "deq called on empty queue"
  | x::tl -> q.contents <- tl; x
  end
end

Here we are using type abstraction to protect the state. Outside of the module, no one knows that queues are implemented with a mutable structure. So, only these functions can modify this structure.
A Better Implementation

• Implementation is slow because of append:
  – q.contents @ [x] copies the entire list each time
  – As the queue gets longer, it takes longer to add data
  – Only has a single reference to the beginning of the list

• Let's do it again with TWO references, one to the beginning (head) and one to the end (tail).
  – Dequeue by updating the head reference (as before)
  – Enqueue by updating the tail of the list

• The list itself must be mutable
  – because we add to one end and remove from the other

• Step 1: Understand the problem
Data Structure for Mutable Queues

```ocaml
type 'a qnode = {
  v: 'a;
  mutable next : 'a qnode option
}

type 'a queue = { mutable head : 'a qnode option;
  mutable tail : 'a qnode option }
```

There are two parts to a mutable queue:

- the “internal nodes” of the queue with links from one to the next
- the head and tail references themselves

All of the references are options so that the queue can be empty.
Queues in the Heap

An empty queue

A queue with one element

A queue with two elements
Visual Shorthand: Abbreviating Options

An empty queue

A queue with one element

A queue with two elements