Programming Languages and Techniques (CIS120)

Lecture 15
February 17, 2016

Mutable Queues
Lecture notes: Chapter 16
What answer does the following expression produce?

```plaintext
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
p2.x <- 42;
p1.x
```

1. 17
2. 42
3. 0
4. runtime error

Answer: 42
let p1 : point = \{x=1; y=1;\}
let p2 : point = p1
let ans : int =
  p2.x <- 17; p1.x

Allocate a Record

Workspace

Stack

Heap
Allocate a Record

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

```plaintext
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.
Look Up ‘p2’

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

Workspace

Stack

Heap

CIS120
let ans : int =
  .x <- 17; p1.x
let ans : int = .x <- 17; p1.x
let ans : int = ()
; p1.x

This is the step in which the ‘imperative’ update occurs. The mutable field x has been modified in place to contain the value 17.
let ans : int = 0; p1.x
let ans : int = p1.x
let ans : int = .x
let ans : int = 
    .x
let ans : int = 17
Let expression

```
let ans : int = 17
```
Push ans

Workspace

Stack
- p1
- p2
- ans 17

Heap
- x 17
- y 1

DONE!
What do the **Stack** and **Heap** look like after simplifying the following code on the workspace?

```ocaml
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
let z = p1.x in
p2.x <- 42;
p1.x
```

**Stack**

1. 

**Heap**

1. 

---

**Answer:** 1
Reference and Equality

= vs. ==
Reference Equality

• 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 object in the heap; so:

\[
\begin{align*}
  r2 & \quad == \quad r3 \\
  \text{not} \quad (r1 & \quad == \quad r2) \\
  r1 & \quad = \quad r2
\end{align*}
\]
Structural vs. Reference Equality

• **Structural (in)equality**: \( v_1 = v_2 \quad v_1 \not= v_2 \)
  
  – 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)
  
  – appropriate for comparing *immutable* datatypes

• **Reference (in)equality**: \( v_1 == v_2 \quad v_1 != v_2 \)
  
  – Only looks at where the two references point in the heap
  
  – function values are only equal to themselves
  
  – equates strictly fewer things than structural equality
  
  – appropriate for comparing *mutable* datatypes
What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
p1 = p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true
What is the result of evaluating the following expression?

```ocaml
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
p1 == p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true
What is the result of evaluating the following expression?

```plaintext
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in
p1 == p2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: false
What is the result of evaluating the following expression?

```ocaml
let p1 : point = { x = 0; y = 0; } in
let p2 : point = { x = 0; y = 0; } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in
l1 = l2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: true
What is the result of evaluating the following expression?

```plaintext
let p1 : point = { x = 0; y = 0; } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false
3. runtime error
4. compile-time error

Answer: false
Putting State to Work

Mutable Queues
Announcements

• HW 4: Mutable Queues is available
  – Due: Tuesday, February 16th at 11:59 pm
Have you ever implemented the mutable data structure called a **linked list**, in any language?

1. yes
2. no
3. not sure
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 data structure 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
  – Both kinds of access must be efficient to handle the volume
(Mutable) Queue Interface

module type QUEUE =
  sig
    (* abstract type *)
    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 end of the queue *)
    val enq : 'a -> 'a queue -> unit

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

We can tell, just looking at this interface, that it is for a MUTABLE data structure. How?

Because queues are mutable, we must allocate a new one every time we need one.

Adding an element to the queue returns unit because it modifies the given queue.
Specify the behavior via test cases

```ocaml
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
```
What value should replace `??` so that the following test passes?

```
let test () : bool =
    let q : int queue = create () in
    enq 1 q;
    let _ = deq q in
    enq 2 q;
    ?? = deq q

;; run_test "enq after deq" test
```

1. 1
2. 2
3. None
4. failwith “empty queue”

Answer: 2
Implementing Linked Queues

Representing links
Implement the behavior

```ocaml
module ListQueue : 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 : 'a 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

- Challenge: The list itself must be mutable
  - because we add to one end and remove from the other
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:

1. the “internal nodes” of the queue, with links from one to the next
2. a record with links to the head and tail nodes

All of the links are *optional* 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 three elements
Given the queue datatype shown below, which expression creates a 1-element queue in the heap:

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

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

1. let q = { head = None; tail = None }

2. let q = { head = 1; tail = None }

3. let q = let qn = { v= 1; next = None } in
   { head = qn; tail = None }

4. let q = let qn = { v= 1; next = None } in
   { head = Some qn; tail = Some qn }

Answer: 4