Programming Languages and Techniques (CIS120)

Lecture 16

Feb 18, 2013

Iteration for Linked Queues
Announcements

• Homework 5 (queues) is on the web
  – It is due Thursday, February 21\textsuperscript{st} at 11:59:59pm

• Exam stats and solutions available on Wednesday
  – One person needs to make-up the make-up
Mutable Queues

singly-linked datastructures
Data Structure for Mutable Queues

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 (and so that the links can terminate).
Linked Queue Invariants

• Just as we imposed some restrictions on which trees are legitimate Binary Search Trees, Linked Queues must also satisfy some invariants:

Either:
(1) head and tail are both None (i.e. the queue is empty)
or
(2) head is Some n1, tail is Some n2 and
   - n2 is reachable from n1 by following ‘next’ pointers
   - n2.next is None

• We can check that these properties rule out all of the “bogus” examples.

• A queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it’s done.
• Suppose we want to extend the interface with a length function:

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

  ...

  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

• How can we implement it?
* Calculate the length of the list using recursion *

```ocaml
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin
      match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- This code for `length` uses a helper function, `loop`:
  - the correctness depends crucially on the queue invariant
  - what happens if we pass in a bogus `q` that is cyclic?

- The height of the ASM stack is proportional to the length of the queue
  - That seems inefficient... why should it take so much space?
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
  loop q.head

let rec loop (no:…) : int =
in
  loop q.head

length q

length q

q
Evaluating length

Workspace

length q

Stack

length

q

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
  loop q.head

head
tail

v 1
next

v 2
next
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
  let rec loop (no:…) : int = …
in
  loop q.head

head

next

v 1

next

v 2

next

v
Evaluating length

Fun (q:'a queue) ->
let rec loop (no:…) : int = …
in
loop q.head

Workspace

Stack

Heap

v 1
next

v 2
next

head
tail
Evaluating length

Workspace

```
let rec loop (no: ...) : int =
  begin
    match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
in
loop q.head
```

Stack

```
length
q
( )
```

Heap

```
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...
in
  loop q.head
```
Evaluating length

Workspace

```ml
let loop = fun (no: ...) ->
  begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
  end
in
loop q.head
```

Stack

```ml
let rec loop (no:...) : int = ...
in
loop q.head
```

Heap

```ml
fun (q:'a queue) ->
  let rec loop (no:...) : int = ...
in
loop q.head
```

```ml
let loop = fun (no: ...) ->
  begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
  end
in
loop q.head
```
Evaluating length

Workspace

```
let loop = fun (no: ...) ->
    begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
    end
    in
    loop q.head
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Heap

```
fun (q:a queue) ->
    let rec loop (no:...) : int = ...
    in
    loop q.head
```

```
<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

| tail   |

<table>
<thead>
<tr>
<th>v 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int = 
  ...
in
  loop q.head

fun (no: ...) ->
begin
  match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

loop q.head

length

q

(head)

q

tail

loop

v 1

next

v 2

next
After a few steps...

(From here on, we’ll take some shortcuts in the ASM animations.)
Evaluating length

Workspace

begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

Stack

fun (q:'a queue) ->
  let rec loop (no:…) : int =
    ...
  in
  loop q.head

Heap

fun (no: …) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
Evaluating length

Workspace:

```
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
```

Stack:

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>
```

Heap:

```
fun (q: 'a queue) ->
  let rec loop (no: ...) : int =
    ...
  in
  loop q.head
```

```
<table>
<thead>
<tr>
<th>v 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>
```

```
| head |
| q    |
| tail |
| loop |
```

```
| ( ) |
```

```
| ( ) |
```

```
| ( ) |
```

```
| ( ) |
```
Evaluating length

Workspace

```
begin match no with
  | None  -> 0
  | Some n -> 1 + (loop n.next)
end
```

Stack

- length
- q

Heap

```
fun (q: 'a queue) ->
  let rec loop (no: ...) : int = ...
in
  loop q.head
```

```
fun (no: ...) ->
  begin match no with
    | None  -> 0
    | Some n -> 1 + (loop n.next)
  end
```
Evaluating length

Workspace

begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

fun (q: 'a queue) ->
let rec loop (no: ...) : int = ...
in loop q.head

begin match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

1 + (loop n.next)

Stack

length

q

( )

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
  loop q.head

head

tail

loop

v 1

next

v 2

next

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

CIS120 / Spring 2013
Evaluating length

Workspace

Stack

Heap

fun (q: 'a queue) ->
let rec loop (no: ...) : int =
  ...
  in
  loop q.head

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

( )

1 + ( )

length

q

loop

v 1

next

no

v 2

next

head

tail

loop q.head

loop q.

q

v

loop q.head

next

no

v
begin match no with
| None    -> 0
| Some n  -> 1 + (loop n.next)
end

fun (q:'a queue) ->
let rec loop (no:…) : int = …
in
loop q.head

fun (no:…) ->
begin match no with
| None    -> 0
| Some n  -> 1 + (loop n.next)
end
...after a few steps...
Evaluating length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
  loop q.head

fun (no: ...) ->
begin match no with
  None -> 0
  Some n -> 1 + (loop n.next)
end

1 + (loop n.next)
...after a few more steps...
begin match no with
| None  -> 0
| Some n -> 1 + (loop n.next)
end
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ... in
    loop q.head

begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

0

Stack

fun (q:'a queue) ->
let rec loop (no:...) : int =
  ... in
  loop q.head

Heap

head

val head : 'a

val tail : 'a

loop

v

val v : int

next

val next : 'a

v

val v : int

next

val next : 'a

1

val v : int

no

val no : 'a

loop

no

val no : 'a

loop

1 + ( )

end

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

1 + ( )

no

val no : 'a

1 + ( )

no

val no : 'a

1 + ( )

no

val no : 'a

1 + ( )

no

val no : 'a

1 + ( )

no

val no : 'a
fun (q:'a queue) ->
let rec loop (no:...) : int =
  ...
in
loop q.head

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

1 + ( )
Evaluating length

Workspace

Stack
- length
- q

Heap
- fun (q:'a queue) ->
  let rec loop (no:…) : int =
  ... in
  loop q.head

1 + 0

1 + ( )

head
- q
- tail
- loop

loop
- v 1
- next
- v 2
- next
- None

no
- no
- n

fun (no: …) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end

fun (no: …) ->
let rec loop (no:…) : int =
  ... in
  loop q.head
fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
  loop q.head

fun (no: …) ->
begin
match no with
| None -> 0
| Some n -> 1 + (loop n.next)
end
fun (q:'a queue) ->
  let rec loop (no:...) : int =
    ...
in
    loop q.head

fun (no: ...) ->
  begin match no with
    | None -> 0
    | Some n -> 1 + (loop n.next)
  end
fun (q:'a queue) ->
let rec loop (no:…) : int =
  ...
in
  loop q.head

fun (no: …) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
fun (q:'a queue) ->
let rec loop (no:...) : int =
    ... in
    loop q.head

fun (no: ...) ->
begin match no with
  | None -> 0
  | Some n -> 1 + (loop n.next)
end
Evaluating length

Workspace

2

Stack

| length | q |

Heap

fun (q:'a queue) ->
let rec loop (no:…) : int =
    ...
in
loop q.head

head
tail

v 1
next

v 2
next

None

fun (no: …) ->
begin match no with
    None -> 0
    Some n -> 1 + (loop n.next)
end

DONE!
Iteration

loops
length (using iteration)

(* Calculate the length of the list using iteration *)

\[
\begin{aligned}
&\text{let } \text{length} (q:\text{'a queue}) : \text{int} = \\
&\quad \text{let rec } \text{loop} (\text{no}:'a qnode \text{ option}) (\text{len}:\text{int}) : \text{int} = \\
&\quad \quad \begin{aligned}
&\text{begin } \text{match} \text{no} \text{ with} \\
&\quad | \text{None} \to \text{len} \\
&\quad | \text{Some} \ n \to \text{loop} \ n.\text{next} \ (1+\text{len}) \\
&\text{end}
\end{aligned} \\
&\quad \text{in} \\
&\quad \text{loop} \ q.\text{head} \ 0
\end{aligned}
\]

- This code for \text{length} also uses a helper function, \text{loop}:
  - This loop takes an extra argument, \text{len}, called the \textit{accumulator}
  - Unlike the previous solution, the computation happens “on the way down” as opposed to “on the way back up”
  - Note that \text{loop} will always be called in an empty workspace—the results of the call to \text{loop} never need to be used to compute another expression. In contrast, we had \((1 + (\text{loop} \ldots))\) in the recursive version.
Tail Call Optimization

• Why does it matter that ‘loop’ is only called in an empty workspace?

• We can *optimize* the abstract stack machine:
  – The workspace pushed onto the stack tells us “what to do” when the function call returns.
  – If the pushed workspace is empty, we will always ‘pop’ immediately after the function call returns.
  – Therefore, we do not need to save the ‘empty’ workspace on the stack.
  – Moreover, any local variables that were pushed so that the current workspace could evaluate will no longer be needed, so we can eagerly pop them too.

• The end result is that we have turned *recursion* into a true *loop*. (Just like a ‘while’ or ‘for’ loop in Java or C.)
Tail Calls and Iterative length

Workspace

length q

Stack

length

q

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  …
in
loop q.head 0

Bindings above this line are top-level declarations.
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
  in
  loop q.head 0

Tail Call!
Tail Calls and Iterative length

Workspace

```
let rec loop (no:'a qnode option) (len:int) : int =
  begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end
in
loop q.head 0
```

Stack

```
<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>
```

Heap

```
fun (q:'a queue) ->
  let rec loop (no:...) (len:int) =
    ...
in
  loop q.head 0
```

```
head

tail

v 1
next

v 2
next
```

Note:

1. No workspace is saved – there is no need do to that for tail calls
2. We pop all the locals (up to the last saved workspace). In this case, there are none.
A detail we’ve been sweeping under the rug until now:

The *closure* of the local recursive function `loop` includes a binding for the loop function itself!

Why? The loop body mentions the loop identifier.
Tail Calls and Iterative length

Workspace: (0)

Stack:
- length
- q

Heap:
- head
- tail
- loop

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next
         (1+len)
  end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin
    match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end

Stack

fun (q:'a queue) ->
  let rec loop (no:…) (len:int)=
    in
    loop q.head 0

Heap

Notes:
- no workspace is saved on the stack
- pop the old locals (q and loop)
- push the closure and argument bindings
- the new workspace is just the body of the function

This binding comes from the closure.
Tail Calls and Iterative length

Workspace

begin match with
  | None -> len
  | Some n -> loop n.next (1+len)
end

Stack

length

q

Heap

fun (q: 'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

loop

head

tail

len 0

v 1

next

v 2

next

loop

fun (no: 'a qnode option) (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

begin match with
| None  -->  len
| Some n  -->  loop n.next (1+len)
end

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Heap

fun (q:'a queue) -->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail</td>
</tr>
<tr>
<td>len</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>len 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>v 1</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>

| v 2   |
| next  |
| loop  |

fun (no:'a qnode option)
(len:int)  -->  begin match no with
  | None  -->  len
  | Some n  -->  loop n.next (1+len)
end
begin match with
   None -> len
   Some n -> loop n.next (1+len)
end

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
   ...
in
   loop q.head 0
Tail Calls and Iterative length

Workspace

```plaintext
(let n.next (1+len))
```

Stack

```
length
q
loop
no
len 0
n
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:…) (len:int)= …
in
loop q.head 0
```

```
head
tail
len
```

```
v 1
next
```

```
v 2
next
```

```
loop
```

```
fun (no:'a qnode option) (len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end
```
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)= ...
in
loop q.head 0

fun (no:'a qnode option)
(len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end

(n.next (1+len))
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  ...
  in
  loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end

CIS120 / Spring 2013
Tail Calls and Iterative length

Workspace

( (1+0) )

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

Head

fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
  ...
  in
  loop q.head 0

fun (no:'a qnode option) (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

( 1 )

Stack

length

q

loop

no

len 0

Heap

fun (q:'a queue) ->
  let rec loop (no:…) (len:int)=
    ...
    in
    loop q.head 0

head
tail
len 0

v 1

next

v 2

next

loop

fun (no:'a qnode option)
(len:int) -> begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end

Tail Call!
Tail Calls and Iterative length

Workspace

begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
end

Stack

length

q

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
    ...
in
loop q.head 0

loop

no

len 1

v 1

next

v 2

next

loop

head
tail

Note: we popped the old values of loop, no, len, and n when we did the tail call. Then we pushed the new values of loop, no, and len.

This leaves the stack in almost the same shape as when we first called loop.

Effectively, we have updated the stack slots for no and len.
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  in
  loop q.head 0

begin match __ with
  None -> len
  | Some n -> loop n.next (1+len)
end

let rec loop (no:…) (len:int)=
  …
in
loop q.head 0

fun (no:'a qnode option)
(len:int) -> begin match no with
  None -> len
  | Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

Stack

Heap

begin match with
| None -> len
| Some n -> loop n.next (1+len)
end

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

fun (no:'a qnode option)
(len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

begin match with
  None -> len
  Some n -> loop n.next (1+len)
end

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
</tr>
<tr>
<td>len 1</td>
</tr>
</tbody>
</table>

Heap

fun (q:'a queue) ->
  let rec loop (no:…) (len:int)=
    …
  in
  loop q.head 0

fun (no:'a qnode option)
  (len:int) -> begin match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end

CIS120 / Spring 2013
Tail Calls and Iterative length

Workspace

```
(  n.next (1+len))
```

Stack

```
| length |
| q |
| loop |
| no |
| len 1 |
| n |
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0
```

```
| head |
| tail |
| len 1 |
| n |

```

```
| v 1 |
| next |
| v 2 |
| next |

```

```
| loop |
| None |
```

```
fun (no:'a qnode option) (len:int) -> begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
```
Tail Calls and Iterative length

Workspace

( (1+len))

Stack

length

q

loop

Heap

fun (q:'a queue) ->

let rec loop (no:...) (len:int)=

...in

loop q.head 0

head
tail

len 1

v 1

next

v 2

next

None

loop

fun (no:'a qnode option) (len:int) -> begin match no with

| None -> len

| Some n -> loop n.next (1+len)

end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q: 'a queue) ->
  let rec loop (no: ...) (len:int) =
    ...
    in
    loop q.head 0

fun (no: 'a qnode option) (len:int) ->
  begin
    match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  …
in
  loop q.head 0

fun (no:'a qnode option)
  (len:int) ->
  begin
    match no with
    | None -> len
    | Some n -> loop n.next (1+len)
  end
Tail Calls and Iterative length

Workspace

( 2 )

Stack

length

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

Heap

loop

head
tail

len 1

v 1

next

v 2

next

None

fun (no:'a qnode option)
  (len:int) -> begin match no with
    None -> len
    Some n -> loop n.next (1+len)
end

Tail Call!
Note: Again, the tail call leaves the stack as before, but effectively updates the values of no and len.

We may as well call this in-place-update of the stack, even though technically these bindings are not mutable!
Tail Calls and Iterative length

Workspace

```
begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end
```

Stack

```
| length |
| q |
```

Heap

```
fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  ...
in
loop q.head 0
```

```
| head |
| tail |
| len 2 |
```

```
| v 1 |
| next |
```

```
| v 2 |
| next |
```

```
None
```

```
| loop |
```

```
fun (no:'a qnode option)
(len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end
```

CIS120 / Spring 2013
Tail Calls and Iterative length

Workspace

```
begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
```

Stack

```
fun (q:'a queue) ->
  let rec loop (no:...) (len:int)=
      ...
  in
  loop q.head 0
```

Heap

```
let rec loop (no:...) (len:int)=
  ...
```

```
begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
```
Tail Calls and Iterative length

Workspace

Stack

Heap

fun (q:'a queue) ->
let rec loop (no:...) (len:int)=
  ...
in
  loop q.head 0

fun (no:'a node option) (len:int) ->
begin match no with
  None -> len
  Some n -> loop n.next (1+len)
end
Tail Calls and Iterative length

Workspace

2

Stack

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop</th>
</tr>
</thead>
</table>

Heap

fun (q: 'a queue) ->
let rec loop (no: ...) (len:int)=
... in
loop q.head 0

head

no

len 2

v 1

next

v 2

next

loop

None

fun (no: 'a qnode option)
(len:int) -> begin match no with
| None -> len
| Some n -> loop n.next (1+len)
end

CIS120 / Spring 2013
Tail Calls and Iterative length

Workspace

2

Stack

length
q

Heap

fun (q:'a queue) ->
let rec loop (no:…) (len:int)=
  ...
  in
  loop q.head 0

fun (no:'a node option)
  (len:int) -> begin match no with
  | None -> len
  | Some n -> loop n.next
    (1+len)
  end

DONE!
Some Observations

- Tail call optimization lets the stack take only a fixed amount of space.
- The “recursive” call to loop (effectively) updates some of the stack bindings in place.
  - We can think of these bindings as the state being modified by each iteration of the loop.

- These two properties are the essence of iteration.
  - They are the difference between recursion and iteration
  - Most imperative programs provide iteration with “while” “for”, and the related “break” and “continue” operations.
  - Tail recursion generalizes all of these.
Infinite Loops

This program will go into an infinite loop.

Unlike a recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will “silently diverge” and never produce an answer...
More iteration examples

to_list
print
get_tail
to_list (using iteration)

(* Retrieve the list of values stored in the queue, ordered from head to tail. *)

let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
    begin
      match no with
      | None -> List.rev l
      | Some n -> loop n.next (n.v::l)
    end
  in loop q.head []

- Here, the state maintained across each iteration of the loop is the queue “index pointer” no and the (reversed) list of elements traversed.
- The “exit case” post processes the list by reversing it.
print (using iteration)

```
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
    | None  -> ()
    | Some n -> print_endline (string_of_element n.v);
       loop n.next
    end
  in
  print_endline "--- queue contents ---";
  loop q.head;
  print_endline "--- end of queue ------"
```

• Here, the only state needed is the queue “index pointer”.
Checking Queue validity

Detecting Loops
Linked Queue Invariants

• Just as we imposed some restrictions on which trees are legitimate Binary Search Trees, Linked Queues must also satisfy some invariants:

Either:
(1) head and tail are both None (i.e. the queue is empty)

or
(2) head is Some n1, tail is Some n2 and
   - n2 is reachable from n1 by following ‘next’ pointers
   - n2.next is None

• We can check that these properties rule out all of the “bogus” examples.

• A queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it’s done.
let valid (q: 'a queue) : bool =
  begin match (q.head, q.tail) with
    | (None, None) -> true
    | (Some(qh), Some(qt)) ->
      begin match get_tail qh with
        | Some n -> qt == n (* tail is the last node *)
        | None -> false
      end
    | (_, _) -> false
  end
get_tail (using iteration)

/* get the tail (if any) from a queue */
let rec get_tail (q: 'a queue) : 'a qnode option =
  let rec loop (qn: 'a qnode) (seen: 'a qnode list)
   : 'a qnode option =
    begin match qn.next with
    | None -> Some qn
    | Some n ->
      if contains_alias n seen then None
      else loop n (qn::seen)
    end
  in loop q.head []

• This function does not assume that q has no cycles.
  – It returns Some n if n is a “tail” reachable from q.head
  – It returns None if there is a cycle in the queue
• The state is an index pointer and a list of all the nodes seen.
  – contains_alias is a helper function that checks to see whether n has an alias in the list