Programming Languages and Techniques (CIS120)

Lecture 13
February 12th, 2016

ASM
Chapter 14
Announcements

• Midterm 1 will be Tuesday evening
  – ROOMS announced on Monday
  – TIME: 6:15PM
  – Covers up to Feb 10\textsuperscript{th} and HW 3
    • no options, records, or Abstract Stack Machine!

• Review session Sunday, Feb 14\textsuperscript{th}  6-8PM in Towne 100
Has this situation ever happened to you?

1. yes

2. no
Have you used the substitution model to reason about how functions evaluate?

1. yes, every single step
2. yes, but skipping some steps
3. no, it seems useless to me
4. what is the substitution model?

```plaintext
let filter (f : 'a -> bool) (l : 'a list) : 'a list =
    begin
      match l with
      | [] -> []
      | hd :: tl ->
        if f hd then hd :: filter f tl
        else filter f tl
    end
```
Modeling (Stateful) Computation
Models of Computation

- Explain the behavior of your program.
  - i.e. the *meaning* or *semantics*

Substitution model works for pure functional programs...

...but:

- How do we *implement* the substitution model?
- How do we explain behaviors like:
  - Stack overflow during evaluation (looping recursion?).

- Where do the lists and trees we construct live in memory?
Towards Imperative Programs

- What about program features that update state:
- E.g. in Java

```java
int x = 3;
x = x + 1;  // what does this do?
```

```java
int x = 3;
int y = x;
y = y + 1;
```

vs.

```java
DataObj x = new DataObj();
DataObj y = x;
y.field = y.field + 1;
```

- Other features: Exceptions, Dynamic Dispatch, Threads, etc.
Mutable Records

• **Mutable** (updateable) state means that the *locations* of values becomes important.

```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
```

• The simple substitution model of program evaluation breaks down – it doesn’t account for locations

• We need to refine our model of how to understand programs.
The Abstract Stack Machine
Stack Machine

• Three “spaces”
  – workspace
    • the expression the computer is currently working with
  – stack
    • temporary storage for let bindings and partially simplified expressions
  – heap
    • storage area for large data structures

• Initial state:
  – workspace contains whole program
  – stack and heap are empty

• Machine operation:
  – In each step, choose next part of the workspace expression and simplify it
  – Stop when there are no more simplifications
Abstract Stack Machine

The abstract stack machine operates by simplifying the expression in the workspace...

... but instead of substitution, it records the values of variables on the stack...
... values themselves are divided into primitive values (also on the stack) and reference values (on the heap).

For immutable structures, this model is just a complicated way of doing substitution
... but we need the extra complexity to understand mutable state.

We'll go through examples here, read Chapter 14 of the lecture notes for general rules
A value is either:

- a primitive value like an integer, or,
- a reference to a location in the heap

A reference is the address of a piece of data in the heap. We draw a reference value as an “arrow”:

- The start of the arrow is the reference itself (i.e. the address).
- The arrow “points” to the value located at the reference’s address.
References as an Abstraction

• In a real computer, the memory consists of an array of 32-bit words, numbered 0 \ldots 2^{32}-1 (for a 32-bit machine)
  – A reference is just an address that tells you where to look up a value
  – Data structures are usually laid out in contiguous blocks of memory
  – Constructor tags are just numbers chosen by the compiler
    e.g. Nil = 42 and Cons = 120120120

```

The “real” heap.

<table>
<thead>
<tr>
<th>Addresses</th>
<th>32-bit Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>4294967291</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4294967290</td>
<td>...</td>
</tr>
<tr>
<td>4294967291</td>
<td>120120120</td>
</tr>
<tr>
<td>4294967292</td>
<td>3</td>
</tr>
<tr>
<td>4294967293</td>
<td>4294967295</td>
</tr>
<tr>
<td>4294967294</td>
<td>...</td>
</tr>
<tr>
<td>4294967295</td>
<td>42</td>
</tr>
</tbody>
</table>

How we picture it.
```

CIS120
The ASM:
let, variables, operators, and if expressions
let x = 10 + 12 in
let y = 2 + x in
if x > 23 then 3 else 4
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
let $x = 22$ in
let $y = 2 + x$ in
if $x > 23$ then 3 else 4
Simplification

Workspace

```plaintext
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```
let y = 2 + x in if x > 23 then 3 else 4
Simplification

Workspace

let \( y = 2 + x \) in
if \( x > 23 \) then 3 else 4

Stack

Heap

\( x \)
\( 22 \)

x is not a value: so look it up in the stack
let y = 2 + 22 in
  if x > 23 then 3 else 4
Simplification

Workspace

let y = \(2 + 22\) in
if \(x > 23\) then 3 else 4

Stack

Heap
Simplification

Workspace

```plaintext
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

| x   | 22 |

Heap
let \( y = 24 \) in
if \( x > 23 \) then 3 else 4
Simplification

Workspace

if \( x > 23 \) then 3 else 4

Stack

\[
\begin{array}{c|c}
    \text{x} & 22 \\
    \text{y} & 24 \\
\end{array}
\]

Heap
Looking up $x$ in the stack proceed from most recent entries to the least recent entries – the “top” (most recent part) of the stack is toward the bottom of the diagram.
Simplification

Workspace

if 22 > 23 then 3 else 4

Stack

x 22
y 24

Heap
Simplification

if 22 > 23 then 3 else 4
Simplification

Workspace

if false then 3 else 4

Stack

x 22
y 24

Heap
Simplification

Workspace

if false then 3 else 4

Stack

x  22
y  24

Heap
Simplification

Workspace

4

Stack

x  22

y  24

Heap

DONE!
What does the **Stack** look like after simplifying the following code on the workspace?

```
let z = 20 in
let w = 2 + z in
  w
```

1. Stack:
   - z 22
   - w 2 + z

2. Stack:
   - z 20
   - w 22

3. Stack:
   - w 22

4. Stack:
   - w 22
   - z 20
What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in
let z = 2 + z in
z
```

1. Stack:
   
   | z | 20 |
   |----|
   | z | 22 |

2. Stack:
   
   | z | 20 |
   |----|
   | z | 22 |

3. Stack:
   
   | z | 22 |
   |----|
   | z | 22 |

4. Stack:
   
   | z | 22 |
   |----|
   | z | 22 |
Simplifying lists and datatypes using the heap
For uniformity, we’ll pretend lists are declared like this:

```ocaml
type 'a list =  
  | Nil  
  | Cons of 'a * 'a list
```
For uniformity, we’ll pretend lists are declared like this:

type `a list =
  | Nil
  | Cons of `a * `a list
Simplification

Workspace

Cons (1, Cons (2, Cons (3, Nil)))

Stack

Heap
Simplification

Workspace

Cons (1, Cons (2, Cons (3, nil)))

Stack

Heap

Nil
Simplification

Workspace

Cons (1, Cons (2, Cons (3, Nil)))

Stack

Heap

Nil
Simplification

Workspace

Cons (1, Cons (2, nil))

Stack

Heap

Nil

Cons 3
Simplification

Workspace
Cons (1, Cons (2, Nil))

Stack

Heap
Nil
Cons 3
Simplification

Workspace
Cons (1, )

Stack

Heap
Cons 3
Cons 2
Nil
Simplification

Workspace

Stack

Heap

Nil

Cons 3

Cons 2

Cons 1

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

```haskell
let z = Cons (1, Nil) in
let w = Cons (2, z) in
w
```

1. Stack
   - `z` (Cons 1
     - nil)
   - `w` (Cons 2
     - Cons 1
       - nil)

2. Stack
   - `z` (Cons 1
     - nil)
   - `w` (Cons 2
     - Cons 1
       - nil)
   - `Nil`
Simplifying functions
let add1 (x : int) : int = x + 1 in add1 (add1 0)
Function Simplification

```latex
let add1 (x : int) : int =
  x + 1
in
add1 (add1 0)
```
let add1 : int -> int =
  fun (x:int) -> x + 1
in
add1 (add1 0)
let add1 : int -> int = fun (x:int) -> x + 1 in add1 (add1 0)
let add1 = in
  add1 (add1 0)

fun (x:int) -> x + 1
Function Simplification

Workspace

```
let add1 = in
  add1 (add1 0)
```

Stack

Heap

```
fun (x:int) -> x + 1
```
Function Simplification

Workspace

add1 (add1 0)

Stack

add1

Heap

fun (x:int) -> x + 1
add1 (add1 0)

fun (x:int) -> x + 1
Function Simplification

Workspace

```
add1 ( 0)
```

Stack

```
add1
```

Heap

```
fun (x:int) -> x + 1
```
Function Simplification

Workspace

add1 (___ 0)

Stack

add1

Heap

fun (x:int) -> x + 1
Do the Call, Saving the Workspace

Workspace: x+1

Stack:
- add1
- add1 (___)
- x 0

Heap:
- fun (x:int) -> x + 1

Note the saved workspace and pushed function argument.
- compare with the workspace on the previous slide.
- the name ‘x’ comes from the name in the heap

The new workspace is the body of the function
Function Simplification

Workspace

x+1

Stack

add1

add1 (___)

Heap

fun (x:int) -> x + 1

0

x
Function Simplification

Workspace

0+1

Stack

add1 (___)

Heap

fun (x:int) -> x + 1

CIS120
Function Simplification

Workspace

0+1

Stack

add1

add1 (___)

x 0

Heap

fun (x:int) -> x + 1
Function Simplification

Workspace

1

Stack

add1

add1 (___)

Heap

fun (x:int) -> x + 1

x 0

POP!
Function Simplification

Workspace

add1 1

Stack

add1

Heap

fun (x:int) -> x + 1

See how the ASM restored the saved workspace, replacing its ‘hole’ with the value computed into the old workspace. (Compare with previous slide.)
Function Simplification

Workspace

```
add1 1
```

Stack

```
add1
```

Heap

```
fun (x:int) -> x + 1
```
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

1
CIS120

Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

add1

1
Function Simplification

Workspace

x+1

Stack

add1

x 1

Heap

fun (x:int) -> x + 1

CIS120
Function Simplification

Workspace

x+1

Stack

add1

Heap

fun (x:int) -> x + 1
Function Simplification

Workspace

1+1

Stack

add1

Heap

fun (x:int) -> x + 1

x 1

add1

fun (x:int) -> x + 1

CIS120
Function Simplification

Workspace

1+1

Stack

add1

Heap

fun (x:int) -> x + 1

CIS120
Function Simplification

Workspace

Stack

Heap

fun (x:int) -> x + 1

```
add1
```

```
x
```

```
1
```

2

POP!
Function Simplification

Workspace

Stack

Heap

2

add1

fun (x:int) -> x + 1

DONE!
A function definition “let rec f (x_1:t_1)...(x_n:t_n) = e in body” is always ready.
   – It is simplified by replacing it with “let f = fun (x:t_1)...(x:t_n) = e in body”

A function “fun (x_1:t_1)...(x_n:t_n) = e” is always ready.
   – It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.

A function call is ready if the function and its arguments are all values
   – it is simplified by
     • saving the current workspace contents on the stack
     • adding bindings for the function’s parameter variables (to the actual argument values) to the end of the stack
     • copying the function’s body to the workspace
Function Completion

When the workspace contains just a single value, we pop the stack by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren’t any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.
What is your current level of comfort with the Abstract Stack Machine?

1. got it well under control
2. OK but need to work with it a little more
3. a little puzzled
4. very puzzled
5. very very puzzled :-}
Simplifying pattern matching & recursion
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
    begin match l1 with
    | Nil  -> l2
    | Cons(h, t) -> Cons(h, append t l2)
    end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in

append a b
let rec append (l1: 'a list) (l2: 'a list) : 'a list = begin
match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
  ______ Cons(h, append t l2)
end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
let append = 
  fun (l1: 'a list) 
    (l2: 'a list) -> 
    begin match l1 with 
    | Nil -> l2 
    | Cons(h, t) -> 
      Cons(h, append t l2) 
    end in 
let a = Cons(1, Nil) in 
let b = Cons(2, Cons(3, Nil)) in 
append a b
let append =
  fun (l1: 'a list) (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
  ______ Cons(h, append t l2)
  ______
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b
let append =
  in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
Let Expression

Note that the reference to a function in the heap is a value.
Create a Stack Binding

Workspace

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
Allocate a Nil cell

Workspace

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
Allocate a Nil cell

Workspace

let a = Cons(1, ) in
let b = Cons(2, Cons(3, Nil))
in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil
Allocate a Cons cell

Workspace

let a = Cons(1, ) in
let b = Cons(2, Cons(3, Nil)) in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begins match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
let a = in
let b = Cons(2, Cons(3, Nil))
in
append a b
let a = in
let b = Cons(2, Cons(3, Nil))
in
append a b
Create a Stack Binding

Workspace

let b = Cons(2, Cons(3, Nil))
in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil
Cons 1
append
a
Allocate a Nil cell

Workspace

```
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

```
append
```

Heap

```
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
```
Allocate a Nil cell

Workspace

let b = Cons(2, Cons(3, Nil))
in
append a b

Stack

append
 a

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil
Cons 1
Nil
Allocate a Cons cell

Workspace

let b = Cons(2, Cons(3, ))
in
append a b

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
let b = Cons(2, )
in append a b
let b = Cons(2, )
in
append a b
Allocate a Cons cell

```
let b = in
append a b
```

```
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
   Cons(h, append t l2)
end
```

Workspace
- `let b = in append a b`

Stack
- `append`
- `a`

Heap
- `Nil`
- `Cons 1`
- `Nil`
- `Cons 3`
- `Cons 2`
let b = ..., in append a b

Workspace

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil
Cons 1
Nil
Cons 3
Cons 2
Create a Stack Binding

Workspace

append a b

Stack

append

a

b

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil

Cons 1

Nil

Cons 3

Cons 2
lookup 'append'

workspace: `append a b`

stack:
- append
- a
- b

heap:
- `fun (l1: 'a list) (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
  end`

heap contents:
- `Nil`
- `Cons 1`
- `Nil`
- `Cons 3`
- `Cons 2`
fun (l1: 'a list) (l2: 'a list) ->
begin
  match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
fun (l1: 'a list) (l2: 'a list) ->
    begin match l1 with
        | Nil -> l2
        | Cons(h, t) -> Cons(h, append t l2)
    end
append C120 Nil Cons 1 Cons 2 Cons 3
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
end
fun (l1: 'a list) (l2: 'a list) ->
begin
match l1 with
| Nil -> l2
| Cons(h, t) ->
    Cons(h, append t l2)
end
Cons(h, append t l2)
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end

append

CIS120
Save Workspace; push l1, l2

Workspace

begin match l1 with
  l Nil -> l2
  l Cons(h, t) ->
    Cons(h, append t l2)
end

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
  l Nil -> l2
  l Cons(h, t) ->
    Cons(h, append t l2)
end

CIS120
begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
  |       Cons(h, append t l2)
end

Workspace

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
  Cons(h, append t l2)
end
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end
Simplify the Branch: push h, t

Workspace

Cons(h, append t l2)

Stack

append
a
b

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin
match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil
Cons 1
Nil
l2
Cons 3
Cons 2
h 1
l1
t
Cons(h, append t l2)

Workspace

Stack
append
a
b
(λ)
l1
l2
h 1
t

Heap
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
Lookup ‘h’

Workspace

Cons(1, append t l2)

Stack

append

a

b

\[
\begin{array}{c}
\text{l1} \\
\text{l2} \\
\text{h} \\
\text{t}
\end{array}
\]

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin
match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
end
Lookup ‘append’

Workspace

Cons(1, append t l2)

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
end
Lookup ‘append’

Workspace

Cons(1, ( φ l2))

Stack

append

a

b

Heap

fun (l1: 'a list) (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
  end

Nil

Cons 1

Nil

Cons 3

Cons 2
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end

Workspace
Cons(1, (t l2))

Stack
append
a
b

Heap
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end
Nil
Cons 1
Nil
l1
l2
h 1
Cons 3
Cons 2
Cons 2
Cons 3
Cons 1
Nil
append

fun \((l1: 'a\ list)\ (l2: 'a\ list)\) ->
begin
  match \(l1\) with
  | \textbf{Nil} -> \(l2\)
  | Cons(h, t) ->
    Cons(h, append t l2)
end

append

CIS120
fun \( (l_1: 'a\ list) \) \\
\( (l_2: 'a\ list) \rightarrow \)
\begin{aligned}
&\text{begin match } l_1 \text{ with} \\
&| \text{Nil} \rightarrow l_2 \\
&| \text{Cons}(h, t) \rightarrow \\
&\quad \text{Cons}(h, \text{append } t \ l_2) \\
&\end{aligned}

$$\text{fun} \ (l1: \ 'a \ list) \ (l2: \ 'a \ list) \ ->$$

$$\begin{align*}
\text{begin} & \quad \text{match} \ l1 \ \text{with} \\
\mid & \quad \text{Nil} \ -> \ l2 \\
\mid & \quad \text{Cons}(h, \ t) \ -> \\
& \quad \text{Cons}(h, \ \text{append} \ t \ l2) \\
\text{end}
\end{align*}$$

Workspace: `Cons(1, ( ))`

Stack:
- `append`
- `a`
- `b`

Heap:
- `Nil`
- `Cons 1`
- `Nil`
- `Cons 3`
- `Cons 2`
- `l1`
- `l2`
- `h 1`
- `t`
Do the Function Call

Workspace

Cons(1, (_______))

Stack

append

a

b

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end

Nil

Cons 1

Nil

Cons 3

Cons 2

CIS120
Save the Workspace; push l1, l2

Workspace

begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
end
lookup 'l1'

begin
    match l1 with
    | Nil -> l2
    | Cons(h, t) -> Cons(h, append t l2)
end
begin match \_ \_ with
| Nil -> l2
| Cons(h, t) ->
| Cons(h, append t l2)
end
begin match l1 with
    | Nil -> l2
    | Cons(h, t) ->
    ________Cons(h, append t l2)
  end
Simplify the Branch (nothing to push)
fun (l1: 'a list) (l2: 'a list) -> begin match l1 with | Nil -> l2 | Cons(h, t) -> Cons(h, append t l2) end

Workspace

Stack
append
a
b
(____)
l1
l2
h 1
t
Cons(1,(____))
l1
l2

Heap
Nil
Cons 1
Nil
Cons 3
Cons 2
fun (l1: 'a list) (l2: 'a list) -> 
begin match l1 with 
| Nil -> l2 
| Cons(h, t) -> 
  Cons(h, append t l2) 
end
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end
Done! Pop stack to last Workspace

(Note that the “returned” value fills in the ‘hole’ of the saved workspace…)

Workspace

Cons(1, )

Stack

append

a

b

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Nil

Cons 1

Nil

Cons 3

Cons 2

Cons 1

Nil

l1

l2

Cons 2

h 1

t

l2

h 1

t

l1

Nil

l1

l2

Cons 3
Allocate a Cons cell

Workspace

Cons(1, __)

Stack

append

a

b

Nil

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end
Allocate a Cons cell

Workspace

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

CIS120
Done! Pop stack to last Workspace

Workspace

Stack
- append
- a
- b

Heap
- fun (l1: 'a list) (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
    Cons(h, append t l2)
  end

POP!
fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
  Cons(h, append t l2)
end

Workspace

Stack
append
a
b

Heap

Nil
Cons 1
Nil
Cons 3
Cons 2
Cons 1

DONE!

CIS120
Done! (PHEW!)

Workspace

Stack

Heap

fun (l1: 'a list) (l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) -> Cons(h, append t l2)
end

Note that the answer [1;2;3] has the same heap cells for its tail as the list ‘b’... but, it does not share any cells with ‘a’.
Simplifying Match

• A match expression
  
  ```
  begin match e with
  | pat₁ -> branch₁
  | ...
  | patₙ -> branchₙ
  end
  ```

  is ready if e is a value

  – Note that e will always be a pointer to a constructor cell in the heap
  – This expression is simplified by finding the first pattern \( \text{pat}_i \) that matches the cell and adding new bindings for the pattern variables (to the parts of e that line up) to the end of the stack
  – replacing the whole match expression in the workspace with the corresponding \( \text{branch}_i \)
Did you attend class today?

1. Yes