CSE 341 AB: Section 3

Josh Pollock
joshpoll@cs.uw.edu
OH: Thursdays 4:30pm - 5:30pm
Intros

Introduce yourself to someone new!

- What’s your name?
- How’s your quarter?
- [insert question here]

- Share questions you have about course content.
  - If the other person can answer it, great!
  - If you both don’t know, hold on to it.
Questions?
Agenda (Lots of Cool Stuff!)

- SML Standard Library
- Datatype Polymorphism
- Tracing Functions (For Real!!)

- Higher-Order Functions
  - Returning Functions
  - map, filter, join, bind/flat_map
  - foldr
  - foldl

- Revisiting HW1
  (see the section code)
SML Standard Library

Online Documentation
http://www.standardml.org/Basis/index.html

Helpful Subset
Top-Level http://www.standardml.org/Basis/top-level-chapter.html
List http://www.standardml.org/Basis/list.html
ListPair http://www.standardml.org/Basis/list-pair.html
Real http://www.standardml.org/Basis/real.html
String http://www.standardml.org/Basis/string.html
Datatype Polymorphism

- Last week we saw polymorphic functions that use parametric polymorphism.
- This week we’ll look at polymorphic datatypes.
- We’ve already seen them, but you can make your own, too!
- As with polymorphic functions, type variables in polymorphic datatypes must be substituted consistently.
- Demo!
# Four Kinds of Functions

<table>
<thead>
<tr>
<th>Input</th>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Term</strong></td>
<td>“Normal” Functions</td>
<td>f (x, y) = x + y</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Parametric</td>
<td>Datatype Polymorphism</td>
</tr>
<tr>
<td></td>
<td>Polymorphism (fake syntax)</td>
<td>f ('a) (x) = x : 'a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>datatype 'a list = ...</td>
</tr>
</tbody>
</table>
## Four Kinds of Functions

<table>
<thead>
<tr>
<th>Input</th>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
</table>
| **Term** | “Normal” Functions $f(x, y) = x + y$ | Dependent Types outside course scope :(
| **Type** | Parametric Polymorphism (fake syntax) $f('a)(x) = x : 'a$ | Datatype Polymorphism `datatype 'a list = ...` |
Function Closures
Functions ARE NOT Values

Closures ARE Values
Function Closures

Function closures are the most unique value we’ll see.

- The only value that’s not an expression.
- Store code and bindings.

- Keep pointers to the code and to an environment.
- The environment stores the bindings that weren’t bound by the function.
  - These are called free variables or open bindings.
  - The environment closes the function.
Function Closures Visualized!

Code

```val foo = 17
val x = 1
val bar = ~4
fun f y = x + y
val y = true
val z = 27```
Function Closures Visualized!

Code
val foo = 17
val x = 1
val bar = ~4
fun f y = x + y
val y = true
val z = 27

Environment
<table>
<thead>
<tr>
<th>id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>17</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>foobar</td>
<td>~4</td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>true</td>
</tr>
<tr>
<td>z</td>
<td>27</td>
</tr>
</tbody>
</table>
Function Closures Visualized!

Pointers are hard to draw.

**Code**

```plaintext
val foo = 17
val x = 1
val bar = ~4
fun f y = x + y
val y = true
val z = 27
```

**Environment**

```
<table>
<thead>
<tr>
<th>id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>17</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>foobar</td>
<td>~4</td>
</tr>
<tr>
<td>f</td>
<td>i0</td>
</tr>
<tr>
<td>y</td>
<td>true</td>
</tr>
<tr>
<td>z</td>
<td>27</td>
</tr>
</tbody>
</table>
```

```
(i0)
<table>
<thead>
<tr>
<th>id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
(e0)
```
Function Closures Visualized!

We need to support recursion!

**Code**

```
val foo = 17
val x = 1
val bar = ~4
fun f y = x + y
val y = true
val z = 27
```

**Environment**

<table>
<thead>
<tr>
<th>id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>17</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>~4</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

```
val foo = 17
val x = 1
val bar = ~4
fun f y = x + y
val y = true
val z = 27
```

We need to support recursion!
Tracing Function Closures
Higher-Order Functions
Returning Functions

Demo!
Higher-order functions really give functional programming its “flavor.”

Today we’ll look at higher-order functions for data manipulation.

They separate data manipulation into two parts:

- Structure traversal
- Computation

It’s also easy to write our own structure traversals.

We’re not stuck with a small set like if, while, and for.
map
Solution: map
map

\[
\begin{bmatrix}
  x_0, & x_1, & x_2, & x_3
\end{bmatrix}
\]

\[
\begin{bmatrix}
  f(x_0), & f(x_1), & f(x_2), & f(x_3)
\end{bmatrix}
\]
map

(* types annotated for clarity *)
fun map (f : 'a -> 'b,
       xs : 'a list) : 'b list =
case xs of
   [] => []
| x::xs' => (f x)::(map (f, xs'))
filter
join
bind/flat_map
The One to Rule Them All: foldr

Remember we can think of constructors as abstract functions and values.

Cons : 'a * 'a my_list -> 'a my_list

Nil : 'a my_list

foldr replaces the constructors with functions you choose.
The One to Rule Them All: foldr
The One to Rule Them All: \texttt{foldr}

\[
\text{(op ::)} : \text{'a} \times \text{'a list} \rightarrow \text{'a list}
\]

\[
[] : \text{'a list}
\]

\[
f : \text{'a} \times \text{'b} \rightarrow \text{'b}
\]

\[
z : \text{'b}
\]

(op :: is the prefix version of ::)
foldr

(* types annotated for clarity *)
fun foldr (f : 'a * 'b -> 'b,
          z : 'b,
          xs : 'a list) : 'b =
    case xs of
      [] => z
    | x::xs' => f (x, foldr (f, z, xs'))
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])

1::(foldr (fn (x, acc) => x::acc, [3, 4], [2])
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])
1::(foldr (fn (x, acc) => x::acc, [3, 4], [2]))
1::(2::(foldr (fn (x, acc) => x::acc, [3, 4], [])))
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])

1::(foldr (fn (x, acc) => x::acc, [3, 4], [2]))

1::(2::(foldr (fn (x, acc) => x::acc, [3, 4], [])))

1::(2::[3, 4])
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])

1::(foldr (fn (x, acc) => x::acc, [3, 4], [2]))

1::(2::(foldr (fn (x, acc) => x::acc, [3, 4], [])))

1::(2::[3, 4])

1::[2, 3, 4]
foldr (fn (x, acc) => x::acc, [3, 4], [1, 2])

1::(foldr (fn (x, acc) => x::acc, [3, 4], [2])

1::(2::(foldr (fn (x, acc) => x::acc, [3, 4], []))

1::(2::[3, 4])

1::[2, 3, 4]

[1, 2, 3, 4]
foldl
What about tail recursion?

Reversing and summing are needlessly slow with `foldr`. 
Solution: foldl

```
(* types annotated for clarity *)
fun foldl (f : 'b * 'a -> 'b,
    acc : 'b,
    xs : 'a list) : 'b =
  case xs of
  [] => acc
  | x::xs' => foldl (f, f (acc, x), xs')
```

foldl generalizes the accumulator pattern
foldr goes down then up
foldl only goes up
foldl (fn (acc, x) => x::acc, [], [1, 2, 3])
foldl (fn (acc, x) => x::acc, [], [1, 2, 3])
foldl (fn (acc, x) => x::acc, [1], [2, 3])
foldl (fn (acc, x) => x::acc, [], [1, 2, 3])
foldl (fn (acc, x) => x::acc, [1], [2, 3])
foldl (fn (acc, x) => x::acc, [2, 1], [3])
foldl (fn (acc, x) => x::acc, [], [1, 2, 3])
foldl (fn (acc, x) => x::acc, [1], [2, 3])
foldl (fn (acc, x) => x::acc, [2, 1], [3])
foldl (fn (acc, x) => x::acc, [3, 2, 1], [])
foldl (fn (acc, x) => x::acc, [], [1, 2, 3])
foldl (fn (acc, x) => x::acc, [1], [2, 3])
foldl (fn (acc, x) => x::acc, [2, 1], [3])
foldl (fn (acc, x) => x::acc, [3, 2, 1], [])
[3, 2, 1]
But foldr Is Still “Better”

You should use foldl when you need tail recursion.

BUT...

- You can write foldl in terms of foldr.
  - We may get to this next week.
- foldr generalizes naturally to other datatypes, and foldl does not.
Generalizing `foldr`.

Most common datatypes have a natural version of `foldr`.*

Generalize the types of the datatype constructors.

Match clause → function (or constant if it has no arguments).

See section file for examples.

*It’s called a *catamorphism*. 
Higher-Order Functions Are Difficult But Useful

HO functions allow for (among other things) better separation of concerns.

Today we saw how you to separate traversal strategies and computation.

It will probably require some time for these functions to sink in.

But once they do, they make your code easier to read and write!