Recap: Functions

Two ways of writing function expressions:

1. Anonymous functions:
   \[
   \text{val } \text{fname } = \text{fn } \rightarrow \text{e}
   \]

2. Named functions:
   \[
   \text{fun } \text{fname } \text{x } = \text{e}
   \]

Recap: Function Application

Function value is: \(<\text{code}, \text{Env at definition}>\)
   \[
   = \langle\text{formal, body-exp}, \text{Env}\rangle
   \]
called a “closure”

(\text{e}_1 \text{ e}_2)

"apply" the argument \text{e}_2 to the (function) \text{e}_1

Evaluation:
1. Evaluate \text{e}_1 in \text{current} env to get (function) \text{v}_1
   \[
   \text{v}_1 \text{ is } \langle\text{formal } \text{x} + \text{body } \text{e}, \text{ env } \text{E}\rangle
   \]
2. Evaluate \text{e}_2 in \text{current} env to get (argument) \text{v}_2
3. Evaluate body \text{e} in env \text{E} extended by binding \text{x} to \text{v}_2

We’ve covered a lot so far

- Core ML features
  - Expressions, Values, Types
- How to build complex values
  - Tuples, records, ...
- How to create, use complex types
  - Lists, trees, expressions,...
- Pattern-matching
- How to write recursive functions

Today: Functions taking, returning functions

Example

\[
\text{fun } \text{f } \text{g } = \\
\text{let} \\
\text{val } \text{x } = 0 \\
\text{in} \\
\text{g } 2 \\
\text{end}; \\
\text{val } \text{x } = 100; \\
\text{fun } \text{h } \text{y } = \text{x } + \text{y}; \\
\text{f } \text{h};
\]
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Creating, (Returning) Functions

Returning functions

\[
\text{val } \text{lt} = \text{fn } x \Rightarrow \text{fn } y \Rightarrow x < y;
\]
\[
\text{fun } \text{lt} \ x \ y = x < y;
\]

In general, these two are (mostly) equivalent:

\[
\text{val } f = \text{fn } x_1 \Rightarrow \ldots \Rightarrow \text{fn } x_n \Rightarrow e
\]
\[
\text{fun } f \ x_1 \ldots \ x_n = e
\]

Returning functions

\[
\text{fun } \text{lt} \ x \ y = x < y;
\]

Parameterized “tester”
- Create many similar testers
- Where is this useful?

Returning functions

\[
\text{fun } \text{lt} \ x \ y = x < y;
\]

Parameterized “tester”
- Create many similar testers
- Where is this useful?

Remember this?

\[
\text{fun } \text{sort } (\text{lt}, \text{l}) = \\
\text{case } \text{l} \text{ of} \\
[] \Rightarrow [] \\
(h::t) = \\
\text{let} \\
\text{val } (l, r) = \text{partition } (\text{lt } h, \text{t}) \\
in \\
(s\text{ort } (\text{lt}, l))@h::(\text{sort } (\text{lt}, r)) \\
\text{end};
\]

- Use “tester” to partition list
  - Tester parameterized by qsort “pivot” h

Function Currying

Tuple version: \[
\text{fun } f \ (x_1, \ldots, x_n) = e
\]

Curried version: \[
\text{fun } f \ x_1 \ldots \ x_n = e
\]

Function Currying

Multiple argument functions by returning a function that takes the next argument
- Named after a person (Haskell Curry)

\[
\text{fun } \text{lt} \ x \ y = x < y;
\]

Could have done: \[
\text{fun } \text{lt} \ (x, y) = x < y;
\]

- But then no “testers” possible
- Must pick good order of arguments
Using parameterized testers

```
fun sort (lt, l) =
  case l of
  | [] => []
  | (h::t) =
    let
      val (l,r) = partition (lt h, t)
      in
        (sort (lt,l)) @ h :: (sort (lt,r))
      end;
```

- Takes a tester (and a list) as argument
- Returns a pair: (list passing test, list failing test)
- Can be called with any tester!

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions
- Using, (Taking) Functions

Taking functions

```
fun filter (_, []) = []
  | filter (f, h::t) =
    if (f h)
    then h :: filter (f, t)
    else filter (f, t));
```

```
fun neg f = fn x => not (f x)
fun partition (f, l) = (filter (f, l), filter (neg f, l))
```

- `filter, neg, partition`: higher-order functions
- Take a any tester as argument!

Higher-order functions: `map`

```
fun map f [] = []
  | map f (h::t) = (f h) :: (map f t)
```

Type says it all!

- Applies “f” to each element in input list
- Makes a list of the results

Map examples

```
fun square x = x * x

fun sq_list = map square

fun toUpper s =
  implode (map Char.toUpper (explode s))
```

Another pattern: Accumulation

```
fun max (x, y) = if x > y then x else y;
fun listMax l =
  let
    fun help (cur, []) = cur
    | help (cur, h::t) = help (max (cur, h), t)
    in
    helper (0, l)
  end;
```

```
fun concat l =
  let
    fun help (cur, []) = cur
    | help (cur, h::t) = help (cur ^ h, t)
    in
    helper ("", l)
  end;
```
What's the pattern?

Examples of fold

```scala
val listMax =

val concat =

val multiplier =
```

Examples of fold

```scala
fun $1 = fold (op:,:) [] $1
```

Funcs taking/returning funcs

Identify common computation “patterns”

- Iterate a function over a set, list, tree ...
- Accumulate some value over a collection

Pull out (factor) “common” code:
- Re-use in many different situations

Funcs taking/returning funcs

Higher-order funcs enable modular code
- Each part only needs local information

Data Structure

Client

Uses list

Uses meta-functions:
map, fold, filter
With locally-dependent funs (lt h), square etc.
Without requiring implement.
Details of data structure

Data Structure

Library

list

Provides meta-functions:
map, fold, filter
to traverse, accumulate over lists, trees etc.
Meta-functions don’t need client info (tester? accumulator?)

Funcs taking/returning funcs

Higher-order funcs enable modular code
- Each part only needs local information

Data Structure

Client (sort)

Uses list

Uses meta-functions: filter
with local funs (lt h)
without impl., details of list

Data Structure

Library

list

Provides meta-functions: filter
Meta-functions don’t need client info (tester?)
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Funcs taking/returning funcs

- Higher-order funcs enable modular code
- Each part only needs local information

- Higher-order funcs enable code composition
  - Join simpler functions to get complex functions
    
    ```
    fun compose (f, g) = fn x => f (g x)
    fold compose (fn x => x)
    ```

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

What is the deal with ’a’?

These meta-functions have strange types:

map: 

filter: 

Why?

Polymorphism

- Poly = many, morph = kind
  
  ```
  fun swap (x, y) = (y, x)  
  'a * 'b -> 'b * 'a
  ```

- ’a and ’b are type variables!
- For-all types: 
  
  ```
  For all ’a, ’b: ’a * ’b -> ’b * ’a
  ```

- ’a,’b can be “instantiated” with any type:
  
  ```
  w/ int,string : int * string -> string * int
  w/ char,int list : char * int list -> int list * char
  w/ int->int,bool : (int -> int) * bool -> bool * (int -> int)
  ```

Instantiation at use

map: 

``` 
fun f x = x + 10; 
val fm = map f; 
```

``` 
fun f x = x**
```

``` 
val fm = map f1 ["cat", "dog", "burrito"]; 
```
Polymorphic ML types

- Poly = many, morph = kind
- Possible ML types:
  
  $tv = 'a \mid 'b \mid 'c \mid \ldots$

  $T = \text{int} \mid \text{bool} \mid \text{string} \mid \text{char} \mid \ldots$

  $T_1 \times T_2 \times \ldots T_n \mid T_1 \to T_2 \mid tv$

- Implicit for-all at the “left” of all types
  - Never printed out, or specified

Polymorphism enables Reuse

- Can reuse generic functions:
  
  map : 'a * 'b \to 'b * 'a
  filter : ('a \to \text{bool} * 'a list) \to 'a list
  rev : 'a list \to 'a list
  length : 'a list \to \text{int}
  swap : 'a * 'b \to 'b * 'a

Genericity is everywhere...

- What’s the type of this function?

  ```
  fun sort (lt,l) = 
    case l of 
      [] => []
    | (h::t) = 
        let 
          val (l,r) = partition ((lt h).t)
        in 
          (sort (lt,l))@h:(sort (l,r))
        end;
  ```

Polymorphism enables Reuse

- Can reuse generic functions:
  
  map : 'a * 'b \to 'b * 'a
  filter : ('a \to \text{bool} * 'a list) \to 'a list
  rev : 'a list \to 'a list
  length : 'a list \to \text{int}
  swap : 'a * 'b \to 'b * 'a
  sort : ('a \to 'a \to \text{bool} * 'a list) \to 'a list
  fold: ...

- If function (algorithm) is “independent” of type, can reuse code for all types!

Not just functions...

- Data types are also polymorphic!

  ```
  datatype 'a list =
    Nil 
    | Cons of ('a * 'a list)
  ```

- Type is instantiated for each use:

Datatypes with many type variables

- Multiple type variables

  ```
  datatype ('a,'b) tree =
    Leaf of ('a * 'b)
    | Node of ('a,'b) tree * ('a,'b) tree
  ```

- Type is instantiated for each use:
**Polymorphic Data Structures**
- “Container” data structures independent of type!
- Appropriate type is instantiated at each use:
  - 'a list
  - ('a, 'b) Tree
  - ('a, 'b) Hashbl ...
- Appropriate type instantiated at use
  - No “casting” as in C++/Java
  - Static type checking catches errors early
  - Cannot add “int” key to “string” hashtable
- **Generics**: feature of next versions of Java, C#, VB...

**Polymorphic Types**
- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

**Remember: ML is statically typed**

```
Expressions (Syntax) ------- Values (Semantics)
  |                      |
  v                      v
Compile-time             Types
  “Dynamic”              |
Exec-time
```

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique
eq type for the expression meaningful type for the expr
3. “ML evaluates expression to compute value”
   - Of the same “type” found in step 2

**Polymorphic Type Inference**
- Computing the types of all expressions
  - At compile time : Statically Typed
- Each binding is processed in order
- Types are computed for each binding
  - For expression and variable bound to
- Types used for subsequent bindings

**Polymorphic Type Inference**
- Can have no idea what a function does, but still know its exact type
- A function may never (or sometimes terminate), but will still have a valid type
- Every expression accepted by ML must have a valid inferred type

**Example 1**
```
val x = 2 + 3
val y = Int.toString x
```
Example 2

```plaintext
val x = 2 + 3
val y = Int.toString x
fun inc y = x + y
```

Example 3

```plaintext
fun foo x = 
  let
    val (y,z) = x
  in
    (-y) + z
  end;
```

Example 4

```plaintext
fun cat [] = ""
  | cat (h::t) = h^ (cat t)
```

Example 5

```plaintext
fun map f [] = []
  | map f (h::t) = (map h)::(map f t)
```

Example 6

```plaintext
fun compose (f,g) x = f (g x)
```
Example 7

```haskell
fun fold f b [] = b
| fold f (h::t)=fold f (f (h,b)) t
```