Functional programming: 2+ weeks

- Scheme
  - Gives a strong, language-based foundation for functional programming
  - May be mostly review for some of you
- Some theory
  - Theoretical foundations and issues in functional programming
- ML
  - A modern basis for discussing key issues in functional programming

Tonight: a final set of topics on functional languages

- ML types
  - user-defined datatypes, variant records, recursive types, polymorphic types, exceptions, streams, ...
- Haskell
  - lazy evaluation
    - purely side-effect free, infinite lists
  - type classes for added flexibility in polymorphism

Then, with luck, on to types

ML concrete user-defined datatypes

- Users can define their own (polymorphic) data structures
- Simple example: ML’s version of enumerated types
  - datatype sign = Positive | Zero | Negative;
- Introduces constants
  - Can be used in patterns

Example

- fun signum(x) =
  if x > 0 then positive
  else if x = 0 then Zero
  else Negative;
val signum = fn : int -> sign;
- fun signum_val(Positive) = 1
  | signum_val(Zero) = 0
  | signum_val(Negative) = -1;
val signum_val = fn : sign -> int;
Variant records/tagged unions

- Each component of a datatype declaration can have information
  - constructors act as functions to create values with that tag
  - can be used in patterns to take apart values of a tag

```
datatype Sexpr =
  Nil |
  Integer of int |
  Symbol of string |
  Cons of Sexpr * Sexpr;
```

Example

```
- Nil;
  Nil : Sexpr;
- Integer;
  Integer : int -> Sexpr;
- Symbol;
  Symbol : string -> Sexpr;
- Cons;
  Symbol : Sexpr * Sexpr -> Sexpr;
```

Using datatypes

```
- val wuss = Cons(Integer 3), Cons(Symbol "hi", Nil));
  Cons(Integer 3, Cons(Symbol "hi", Nil)) : Sexpr; (* '(3 hi)   *)
- fun car Nil = Nil |
  val car = fn : Sexpr -> Sexpr;
  fun cdr Nil = Nil |
  val cdr = fn : Sexpr -> Sexpr;
- cdr wuss;
  Cons(Symbol "hi", Nil) : Sexpr;
- car wuss = Integer 3;
  true : bool;
```

Recursive user-defined datatypes

```
- datatype int_tree =
  Empty |
  Node of int * int_tree * int_tree;
- fun insert x Empty = Node(x,Empty,Empty)
  | insert x (n as Node(y,t1,t2)) =
    if x = y then n
    else if x < y then Node(y,insert x t1,t2)
    else Node(y,t1,insert x t2);
  val insert = fn : int -> int_tree -> int_tree;
- fun member x Empty = false |
  member x (Node(y,t1,t2)) =
    if x = y then true
    else if x < y then member x t1
    else member x t2;
  val member = fn : int -> int_tree -> bool;
```

But what about a polymorphic version?

- It should be polymorphic with respect to = and <
- int_tree is an equality type
  - Does = do the right thing?
- Define using explicit type variables

Polymorphic binary trees

```
- datatype 'a tr =
  Empty |
  Node of 'a * 'a tr * 'a tr;
- fun ins eq lt x Empty = Node(x,Empty,Empty)
  | ins eq lt x (n as Node(y,t1,t2)) =
    if eq(x,y) then n
    else if lt(x,y) then
      Node(y,ins eq lt x t1,t2)
    else Node(y,ins eq lt x t1,t2);
  val ins = fn : ('a*'a->bool) -> ('a*'a->bool)
    -> 'a tr -> 'a tr;
```
That’s a mouthful: use a wrapper

```plaintext
- datatype 'a tree =
  | Tree of {'a tr,
  | eq: ('a->'a) -> bool,
  | lt: ('a->'a) -> bool};
- fun make_tree eq_fn lt_fn =
  | Tree{tree=Empty,eq=eq_fn,lt=lt_fn};
- val make_tree = fn : ('a->'a) -> ('a->'a) -> 'a tree;
- fun insert x (Tree {tree=tr,eq_fn=fn,lt_fn=lt}) =
  | Tree{tree=ins eq_fn lt_fn x tr,
  | eq=fn,lt=lt_fn};
- val insert = fn : 'a -> 'a tree -> 'a tree;
```

A problem

- In Scheme we can use
  “distinguished values” to handle exceptional cases
  ```plaintext
  - (define (find pred x)
    (cond ((null? x) #f)
  | (pred (car x)) (car x))
  | (else (find pred (cdr x)))))
  - (find is-positive? (-3 3 5)) => 3
  - (find is-positive? (-3 -5)) => #f
  ```

In ML it doesn’t work

```plaintext
- fun find pred nil     = false
| find pred (x::xs) =
  | if pred x then x else find pred xs;
- val find = fn : (bool->bool) -> bool list -> bool
- find is_positive [-3,3,5];
...type error...
```

Use exceptions

- Exceptions can be returned from functions without affecting the normal return type
  ```plaintext
  - exception NotFound;
  - fun find pred nil     = raise NotFound
  | find pred (x::xs) =
  | if pred x then x else find pred xs;
- val find = fn : ('a->bool) -> 'a list -> 'a
- find is_positive [-3,3,5];
  3 : int
- find is_positive [-3,-5];
  uncaught exception NotFound
  ```

Handling exceptions

- Add handler clause to expressions to handle (some) exceptions raised in that expression
  ```plaintext
  - (find is_positive [-3,5])
  handle NotFound => 0
  0 : int
  ```

Exceptions can have arguments

```plaintext
- exception IOError of int;
- (...raise IOError(-3) ...
  handle IOError(code) => code ...
```
Streams

- Streams are (in essence) infinite lists
- Streams are a good model for I/O (and other things)
  - Unix pipes are basically streams
- But it’s hard to have an infinite list in an eager-evaluation language
  - Think about appending an element to a list
  - First you evaluate the element and the list, and then you append ... whoops!

Streams in ML

- Instead, represent a stream cons cell as a pair of
  - a head value and
  - a function that will return the next element in the stream
- datatype 'a stream = Stream of 'a * (unit -> 'a stream);

Basic functions

- fun cons_stream(x,f) = Stream(x,f);
- fun hd_stream(Stream(x,f)) = x;
- fun tl_stream(Stream(x,f)) = f();
- fun ints_from(x) = cons_stream(x, fn() => ints_from(x+1));
- val nats = ints_from(0);
- fun map_stream(g,s) = cons_stream(g(hd_stream(s)),
  fn() => map_stream(g,tl_stream(s)));
- val squares = map_stream(fn(x)=>x*x,nats);

References

- ML allows side-effects through explicit reference values
  - Completely non-functional
  - ref : 'a -> 'a ref
  - ! : 'a ref -> a
  - (op :=) : 'a ref * 'a -> unit

Examples

- val v = ref 0;
- val v = ref 0 : int ref
- v := !v + 1;
- () : unit
- !v;
- 1 : int

- A major difference from Scheme is that the mutable objects are stated explicitly
  - In Scheme, set! can be used anywhere, anytime

Modules

- datatypes were cool, but they exposed their representation
  - Helped with pattern matching, etc.
- ML modules support encapsulated abstract data types
  - hidden operations, values, types, and some kinds of polymorphism
Note

- The module system in ML is clearly intended to try to make the language more industrial strength and feasible for practical use.
- A challenge is balancing the software engineering needs with the type system in ML.

Overview

- Structure defines module implementation.
- Signature defines module interface.
  - Hides other aspects of underlying structure.
- Open imports a module for naming convenience.
  - We won’t cover this.
- Functor supports parameterized module implementations.

Structures

- Package a set of declarations.
  
  ```ml
  structure Queue1 = struct
  type 'a T = 'a list; (* T is conventional name *)
  (* constructors *)
  val empty = nil;
  fun enq x q = a @ [x]; (* @ is append *)
  (* accessors *)
  exception empty_queue;
  fun head (x::q) = x
  | head nil = raise empty_queue;
  fun deq (x::q) = (x,q)
  | deq nil = raise empty_queue;
  end;
  ```

Accessing members

- val q = Queue1.enq 3 Queue1.empty;
  val q = [3] : int list
- val q2 = Queue1.enq 4 q;
  val q2 = [3,4] : int list
- Queue1.head q2;
  3 : int

Signatures

- Construct for encapsulating representations.
- Define a public external interface with signature.
- Then apply the signature to restrict the interface to a structure.

Example

```ml
signature QUEUE = sig
  type 'a T;
  val empty : 'a T;
  val enq: 'a -> 'a T -> 'a T;
  exception empty_queue;
  val head: 'a T -> 'a;
  val deq: 'a T -> 'a * 'a T;
end;
structure Queue2: QUEUE = struct .. end;
```

- Any operations in struct that aren’t in sig are inaccessible.
Holes in encapsulation

- Signatures don't completely hide module implementation
- Types defined using type are not hidden
  - Queue.empty = nil;
  - true : bool;
- If you want to hide types, use datatype instead of type

Another hole

- Built-in equality (=) function operates over the representation, not the abstraction
- That is, two values that are abstractly the same can be revealed to be different using =
- There are various proposals to try to fix these holes in ML

Aside: abstract/concrete data

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Abstract</th>
<th>Abstract</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>Operation</td>
<td>Representation</td>
<td>Operation</td>
</tr>
<tr>
<td>Concrete</td>
<td>Concrete</td>
<td>Concrete</td>
<td>Concrete</td>
</tr>
</tbody>
</table>

Functors

- You can parameterize structures by other structures
- Then instantiate the functors to build regular structures
- This only knows the aspects of Q that are defined by QUEUE

Example

```
signature ORDERED = sig
  type T;
  val eq: T * T -> bool;
  val lt: T * T -> bool;
end;
functor Sort(O:ORDERED) = struct
  fun min(x,y) =
    if O.lt(x,y) then x else y;
  fun sort(lst) = ... O.lt(x,y) ...
end;
```

Example con’t

```
structure IntOrder = struct
  type T = int;
  val eq = (op =);
  val lt = (op <);
end;
structure IntSort = Sort(IntOrder);
IntSort.sort([3,5,-2]);
```
Signature “subtyping”

- A quick preview of one of the Cardelli-Wegner ideas
- How can we have subtyping in a language that doesn’t even have inheritance?
- The question is: under what conditions can we treat an instance of one type as an instance of another type?
- Roughly: if all possible instances of type S can be treated as instances of type T, then we can view S as a subtype of T

In ML

- A signature defines a particular interface
- Any structure that satisfies that interface can be used where that interface is expected
  - For instance, in a functor application
- A structure can have more than is required by the signature
  - More operations, more general/polymorphic operations, more details of implementation of the types

Limitations in ML

- Structures and signatures are not first-class values
  - They must be named
  - They must be declared at the top-level or nested inside another structure or signature
- You cannot instantiate functors at runtime to create “objects”
  - This implies you cannot simulate classes and object-oriented programming

Modules vs. ADTs in ML

- ML abstract data types implicitly define a single type
  - With associated constructors, observers and mutators
- Modules can define 0, 1 or many types in the same module with associated operations over several types
  - Multiple types can share private data and operations
- Functors are similar to parameterized ADTs
- Modules are more general, but clumsier for the common case

Haskell

- A “competitor” to ML
- We won’t do a full language description, but will focus on “interesting” differences
  - Lazy evaluation instead of eager
    - Purely side-effect-free
  - Type classes for more flexible polymorphic type checking
  - Unparameterized modules

A bit of history

- Main design completed in 1992
  - By committee
- Attempted to merge the many different lazy-evaluation-based functional languages into one common thrust
  - Miranda, HOPE, …
A few quick, minor examples

map f [] = []
map f (x:xs) = f x : map f xs
<<fn>> :: (a->b) -> [a] -> [b]
let = map square [3,4,5]
[9,16,25] :: [Int]
(3,4,\y -> x+y)
(3,4,<fn>) :: (Int,Int,Int->Int,Int)

List comprehensions

● A nice syntax for constructing lists from generators and guards

- [ expr | var <- expr, ..., ... boolExpr, ...]
  [ f x | x <- xs ]
  [(x,y) | x <- xs, y <- ys]
  [ y | y <- ys, y > 10 ]

quicksort

quicksort [] = []
quicksort (x:xs) =
  quicksort [y | y <- xs, y < x] ++
  [x] ++
  quicksort [y | y <- xs, y >= x]

Easy to construct arithmetic sequences

● [1..8] -- [1,2,3,4,5,6,7,8]
● [2,4..8] -- [2,4,6,8]
● [2,4..] -- [2,4,6,8,10,12,...]
● [1..] -- [1,2,3,4,5,6,7,...]

Sections

● Can call an infix operator on 0 or 1 of its arguments to create a curried function
  (+)
  <<fn>> :: Int -> Int -> Int
  (+ 1) -- increment function
  <<fn>> :: Int -> Int
  (0 _) -- negate function
  <<fn>> :: Int -> Int

Lazy vs. eager evaluation

● Eager, applicative-order, strict
  - Before passing value to function
● Lazy, normal-order, nonstrict, call-by-need, demand-driven
  - When/if first needed
● Again, Haskell is lazy
**Example**

```haskell
my_if test then_val else_val =  
  if test then then_val else else_val

my_if True 3 4  
3 : Int

my_if False 3 4  
4 : Int

y = 12
my_if (x /= 0) (y `div` x) (-1)  
4 : Int
```

Different than in Scheme and ML, which would require a special form

---

**Streams in Haskell**

- All lists are automatically streams!
  - head, tail fields of a list structure won’t be evaluated until they are demanded by some client of the list
- Lazy evaluation holds for all data structures in the same way

---

**Examples**

```haskell
ints_from n = n : ints_from (n+1)  
--same as [n..]
nats = ints_from 0;  
squares = map (^2) nats  
[0,1,4,9,16,25..]  
  
fibs = 0 : 1 :  
    [a+b | (a,b) <- zip fibs (tail fibs)]  
  
[0,1,1,2,3,5,8,13,21,34,55,..]
```

---

**Lazy programming paradigm**

- There is a programming style that exploits lazy evaluation
  - May lead to more reusable components
- Construct a toolkit of operations to generate interesting streams
  - Ex: Scanner produces a stream of tokens
  - Ex: Input produces a stream of characters
  - Ex: Event-driven simulations produce streams of events
- Independently produce operations to manipulate and extract the interesting subset of the generated streams

---

**Polymorphic functions**

- ML allows functions to be
  - Completely polymorphic
  - Polymorphic over types that admit =
    - `eq_pair`: `(``a``*``b`)*(``a``*``b`) -> `bool`
  - Monomorphic
    - `fun square n = n * n`
      - `int` or `real`, but not both
- With the singular exception of equality types, ML supports universal or unbounded parametric polymorphism

---

**Bounded polymorphism**

- It is also possible to support forms of bounded polymorphism, where constraints are expressing on possible instantiating types; examples:
  - polymorphic over all types that support `+`
  - polymorphic over all types that support `+`, `*`
  - polymorphic over all types that support `print`
  - polymorphic over all tuples with at least two components
  - polymorphic over all records with `hd` and `tl` fields
  - ...
More

● Constraints on type parameters let the body know what operations can be performed on expressions of those types
  – Unbounded type values can be passed around, but with no constraints on the operations
● How to express constraints?

Subtype constraints

● In OO languages, we can often express constraints such as “polymorphic over all types that are subtypes of T”
  – subtypes have all the operations of T (and maybe more)
  – body can perform all operations listed in T

Type classes in Haskell

● Haskell supports a similar idea, within a lazy, function, type-inferencing-based language framework
  – Similar to OO classes, but not identical

Example

class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  ● Eq is the name of the new type class
  ● == and /= are the newly declared names of operations on this class
  ● a is the dummy name of a type that’s in this class
    – used in the type signatures of operations of the class
    – roughly like a formal type parameter

Instances of type classes

● Types explicitly declared as members of particular type classes
  – Use instance construct
  – They must provide implementations of the type class’ operations

Instance Eq Int where
  x == y = intEq x y
  x /= y = intNeq x y

Instance Eq Float where
  x == y = floatEq x y
  x /= y = floatNeq x y

Equations

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>y</th>
<th>x == y</th>
<th>x /= y</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3.4</td>
<td>5.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type classes as polymorphic constraints

● Can use a type class to constrain legal instantiations

eq_pair (x1,y1) (x2,y2) = x1==x2 and y1==y2

eq_pair :: (Eq a,Eq b) -> (a,b) -> Bool

● (eq a,Eq b) is a context that constrains the polymorphic type variables a and b to be instances of the Eq class
Defining contexts

- Can be implicitly defined by the type inference system based on operations used in the body
  - Requires that operations are defined in only one class
  - Cannot overload signatures in multiple classes
- Contexts can also be defined explicitly

```haskell
member :: (Eq a) => a -> [a] -> Bool
member _ [] = False
member x (y:ys) = x == y || member x ys
```

Conditional instances

- “A pair supports -- if its component types do”
  
  \[
  \text{instance (Eq a, Eq b) => Eq (a, b) where}
  \text{  \((x_1,y_1) == (x_2,y_2) \iff x_1 == x_2 \land y_1 == y_2\)}
  \text{  \(x /= y \iff \neg (x == y)\)}
  \]

- A list of a supports -- if a does
  
  \[
  \text{instance (Eq a) => Eq [a] where}
  \text{  \([\ ] == [\ ] \iff \text{True}\)}
  \text{  \((x:xs) == (y:ys) \iff x == y \land xs == ys\)}
  \text{  \_ == \_ \iff False\)}
  \]

Default implementations in type classes

- Add a /= operation, which defaults to negation
  
  \[
  \text{- class Eq a where}
  \text{  \(==\), } (/\text{=} : a \rightarrow a \rightarrow \text{Bool}
  \text{  \(x /= y \iff \neg (x == y)\)}
  \]

  \[
  \text{- instance (Eq a, Eq b) => Eq (a, b) where}
  \text{  \((x_1,y_1) == (x_2,y_2) \iff x_1 == x_2 \land y_1 == y_2\)}
  \text{  -- inherits default /=,}
  \text{  -- but could override\)}
  \]

Type subclasses

- Can define new type classes that extend existing type classes, adding new operations and/or defaults
  
  \[
  \text{- Define the superclass(es) as contexts}
  \text{  \bullet Instantiate each of a type's superclasses top-down to satisfy context}
  \text{  \bullet Multiple inheritance allowed}
  \text{  \bullet No name clashes, since operations can not be overloaded\)}
  \]

Hierarchy of predefined type classes in Haskell

Type classes vs. OO subtypes

- Type classes do not support run-time heterogeneous collections
  
  \[
  \text{- Cannot have functions that accept lists of mixed ints and reals}
  \text{  \bullet (Roughly) no run-time subtyping, only compile-time subtyping\)}

- The constraints defined using type classes are not straightforward to define in most OO languages
Type classes vs. ML polymorphism

- ML polymorphism simple with warts
  - equality-bounded polymorphism
  - overloaded operators block some kinds of polymorphism
- Haskell type classes subsume and unify
  unbounded, equality-bounded, and general bounded polymorphism
  - Default implementations are nice, too
- Type classes
  - Big part of standard library and reference manual
  - Temptation is high to go overboard in refining class hierarchy

Whew

- Next week, on to some more discussion of types
- Leading into object-oriented programming languages
- Watch for a new assignment and some readings