Haskell

Many similarities with ML

- strongly typed
- statically typed
  - polymorphic type system
  - automatic type inference
- expression-oriented
- functional
- list-oriented, garbage-collected heap-based
- pattern matching
- exceptions
- modules
- highly regular and expressive

Key differences:

- **lazy evaluation** instead of eager evaluation
  - purely side-effect-free
- **type classes** for more flexible polymorphic typechecking
- unparameterized modules, with interesting encapsulation
- some syntactic clean-ups and conveniences

History

Main design completed in 1992, by a committee

Attempted to merge the many different lazy-evaluation-based functional languages into one common thrust

- Miranda, by Turner [1985]
- HOPE
- ...

Some syntactic differences with ML

ML:

```
- fun map f nil     = nil
    | map f (x::xs) = f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list
- val lst = map square [3,4,5];
[9,16,25] : int list
- (3, 4, fn x y => x+y)
(3,4,fn) : int * int * (int->int->int)
```

Haskell:

```
map f []     = []
map f (x:xs) = f x : map f xs
<<fn>> :: (a->b) -> [a] -> [b]

lst = map square [3,4,5]
[9,16,25] :: [Int]

(3, 4, \x y -> x+y)
(3,4,<<fn>>) : (Int, Int, Int->Int->Int)
```


More examples

ML:

```
- datatype 'a Tree =
    Empty | Node of 'a * 'a Tree * 'a Tree;
- fun size Empty = 0
    | size (Node(_,t1,t2)) = 1+size t1+size t2;
- Node(3,Empty,Empty)
Node(3,Empty,Empty) : int Tree
```

Haskell:

```
data Tree a = Empty | Node a (Tree a) (Tree a)

size Empty = 0
size (Node _ t1 t2) = 1 + size t1 + size t2

Node 3 Empty Empty
... error: cannot find "show" function ...

Node 4
<<fn>> :: Tree Int -> Tree Int -> Tree Int

size (Node 4 (Node 3 Empty Empty) Empty)  → 2
```

General syntactic principles

Expressions and types use similar syntax

- `(3,"hi") :: (Int,String)`
- `[3,4,5] :: [Int]`

Upper-case letters for constructor constants and known types
Lower-case letters for variables and type variables

Functions and variables defined in same way, with no leading keyword

- variables have no arguments
- functions have 1 or more arguments

Layout & indentation are significant, and imply grouping and nesting

- can use `{ ... }` to explicitly control grouping

List comprehensions

Nice syntax for constructing a list from **generators** and **guards**:

`[` *expr* | *var* <- *expr*, ..., *boolExpr*, ... `]`

```
[ f x | x <- xs ]                 -- map f xs
[ (x,y) | x <- xs, y <- ys ]   -- zip xs ys
[ y | y <- ys, y > 10 ]         -- filter (> 10) ys

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

Arithmetic sequences easy to construct, too

```
[1..10]      → [1,2,3,4,5,6,7,8,9,10]
[2,4..10]    → [2,4,6,8,10]
[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 that takes the remaining argument(s)

```
3 + 4
7 :: Int

(+)
<<fn>> :: Int -> Int -> Int

(+ 1)   -- the increment function
<<fn>> :: Int -> Int

(0 -)   -- the negate function
<<fn>> :: Int -> Int
```


Lazy vs. eager evaluation

When is a function argument evaluated?

- eager, applicative-order, strict: before passing value to function
- lazy, normal-order, nonstrict, call-by-need, demand-driven: when/if first needed

When is an expression needed to be evaluated?

- when it's being called as a function
- when it's being used as the test of an if
- when it's being used to pattern-match against (but then only enough to get the constructor tag; the components don't need to be evaluated until they're needed)

Example

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

my_if True 3 4      → 3
my_if False 3 4     → 4

x = 3
y = 12
my_if (x /= 0) (y `div` x) (-1)  → 4
  -- different than in ML or Scheme!
```

A call to `my_if` doesn't evaluate its arguments first

The test is always evaluated, since it's needed to progress

Either the `then_val` or the `else_val` is evaluated, but not both

Needed "special form" in Scheme & ML to achieve this

Needed explicit `lambda`s to get streams in Scheme & ML

Unnecessary in a lazy language

Issues with lazy evaluation

Only computations needed for getting the result need to be evaluated

- can avoid useless work
- can write programs that look inefficient but need not be
  - generator + transformer style

Evaluation order becomes hard to determine
⇒ depends on what caller of function demands

- side-effects become unworkable
- I/O becomes painful
- can always replace variable with defined expression
  ⇒ better equational reasoning

Streams in Haskell

All lists are streams automatically!

- car, cdr fields of a list structure won't be evaluated until they're demanded by some client of the list

Lazy evaluation holds for all data structures in same way

```
ints_from n = n : ints_from (n + 1)
  -- shorthand: [n..]

nats = ints_from 0  -- shorthand: [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, ...
```


Generators and transformers

Programming style exploiting lazy evaluation, leading to more reusable components

Construct a toolkit of operations to generate interesting streams

- lots of list processing functions, usually mapping & filtering & (un)zipping streams
- scanner produces a stream of tokens
- input produces a stream of characters
- event-driven simulations produce streams of events
- ....

Independently produce operations to manipulate and extract interesting **subset** of generated data

- only portion needed in final result will be generated

Example

Implement scanner as a generator of a stream of tokens

- parser is a consumer of the stream
- utility to check change in function compares two streams, only until first difference

Implement parser to produce a stream of possible parses, if grammar has type-dependent ambiguities (like C++)

- typechecker consumes possible parse trees, selects one that typechecks

Polymorphic and overloaded functions

In ML, functions may either be

- completely polymorphic (e.g. `length:'a list→int`) or
- polymorphic over types that admit equality (e.g. `eq_pair:(''a*''b)*(''a*''b)→bool`) or
- completely monomorphic (e.g. `square:int→int`)

Can't define more restricted forms of polymorphism, e.g. a function that is polymorphic over numbers

E.g.

```
fun square n = n * n;
```

requires `n` either to be `int` or `real`, but not either

`*` refers to two different **overloaded** functions, not one **polymorphic** function

- can't define functions polymorphic over the different overloadings

With the one oddball exception of equality types, ML supports **universal** or **unbounded** parametric polymorphism

Bounded polymorphism

Can support forms of **bounded** polymorphism, where can express constraints on possible instantiating types

E.g.:

- polymorphic over all types that support = (equality types)
- polymorphic over all types that support `*` and `+`
- polymorphic over all types that support `print`
- polymorphic over all tuples with at least 2 components
- polymorphic over all records with `hd` and `tl` fields
- ...

Constraints on type parameters let body know what operations can be performed on expressions of those types

- unbounded type variables: can only pass around

How to express constraints?

Subtype constraints

In object-oriented languages, can often express constraints 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*

E.g.

```
class number {
  fun +:(number, number)→number;
  fun *:(number, number)→number;
  ...
};

- fun square n = n * n;
val square = fn : ('a<=number) → ('b<=number);
- square 3;
9 : number
- square 3.4;
11.5 : number
```

[How to get result type to be same as argument?]

Type classes in Haskell

Haskell supports a similar idea, within a lazy, functional, type-inference-based language framework

- similar to OO classes
- some key differences that limit its expressive power

Example: the class `Eq` of types `a` that implement `==`

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

- `Eq` is the name of the new type class
- `==` and `/=` are 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

Instances of type classes

Types must be explicitly declared to be members of particular type classes

- must provide implementations of type class's operations

```
-- Int, Float are previously declared types
instance Eq Int where     -- Int ∈ Eq
  x == y  = intEq x y
  x /= y  = intNeq x y
instance Eq Float where  -- Float ∈ Eq
  x == y  = floatEq x y
  x /= y  = floatNeq x y
```

Now can invoke type class operations on member types

```
3    == 4       -- allowed
3.4  /= 5.6     -- allowed
3    == 4.5     -- type error
"hi" == "there" -- type error
```


Type classes as constraints on polymorphism

Use a type class to constrain legal instantiations

E.g.:

```
eq_pair (x1,y1) (x2,y2)  =  x1==x2 and y1==y2
eq_pair :: (Eq a,Eq b)=> (a,b)→(a,b)→Bool
```

`(Eq a,Eq b)` is a **context**, constraining the polymorphic type variables `a` and `b` to be instances of the `Eq` class

Contexts can be inferred 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 given explicitly (as can regular types)

Another example:

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


Conditional instances

Can use context to place constraints on type variables for when something is a type class instance

"A pair supports `==` if its component types do"

```
instance (Eq a,Eq b)=> Eq (a,b) where
  (x1,y1) == (x2,y2)  =  x1==x2 and y1==y2
  x /= y              =  not (x == y)
```

"A list of `a` supports `==` if `a` does"

```
instance (Eq a)=> Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x==y and xs==ys
  _      == _       =  False
```

Default implementations in type classes

Add a /= operation, which defaults to negating ==

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y  =  not (x == y)
```

Instances can "inherit" this default implementation, or provide their own

```
instance Eq Int where
  x == y  = intEq  x y
  x /= y  = intNeq x y    -- override default

instance (Eq a, Eq b)=> Eq (a,b) where
  (x1,y1) == (x2,y2)  =  x1==x2 and y1==y2
  -- inherit default /=
```


Type subclasses

Can define new type classes that extend existing type classes, adding new operations and/or defaults

- define the superclass(es) as contexts
  - must instantiate each of a type's superclasses top-down to satisfy context
  - subclass = subset
- multiple inheritance allowed
  - name clashes can't happen since operations not overloadable

Example: `Ord` class of ordered things, subclassing `Eq`

```
class (Eq a)=> Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  min, max :: a -> a -> a
  x <= y  =  x == y or x < y
  min x y = if x < y then x else y
  ... (>=, >, and max defaulted too) ...

member_sorted :: (Ord a)=> a -> [a] -> Bool
member_sorted _ []      =  False
member_sorted x (y:ys)  =
   x==y or x<y and member_sorted x ys
```

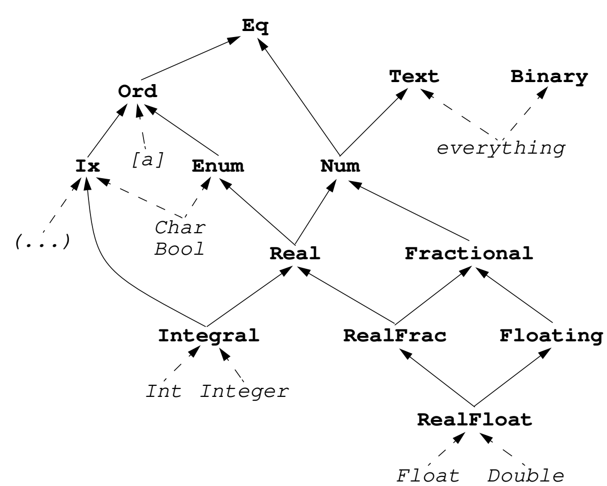

`Ord` instances

```
-- assume instance Eq Int declared already
instance Ord Int where
  x < y   = intLt  x y
  x <= y  = intLeq x y
  ...

-- assume instance Eq (a,b) declared already
instance Ord (a,b) where
  (x1,y1) < (x2,y2) = x1<x2 or x1==x2 and y1<y2
  -- all other operations inherited

-- assume instance Eq [a] declared already
instance Ord [a] where
  []     < (y:ys)   = True
  (x:xs) < (y:ys)   = x<y or x==y and xs<ys
  _      < _        = False
  -- all other operations inherited
```


Hierarchy of predefined type classes in Haskell



Type classes vs. OO subtypes

Type classes do not support run-time heterogeneous collections

- can have functions that are polymorphic over lists of ints and lists of reals
- cannot have functions that accept lists of mixed ints and reals
- no run-time subtyping, just compile-time subtyping (roughly)

Type classes support binary operations like == and + well

```
(==) :: (Eq a)=> a -> a -> Bool
(+)  :: (Num a)=> a -> a -> a
```

Hard to do in OO language without **F-bounded subtype polymorphism** or similar feature

Type classes vs. ML polymorphism

ML polymorphism is simple, but has warts:

- “equality-bounded” polymorphism
- overloaded operators that block polymorphism

Haskell’s type classes subsume and unify unbounded polymorphism, equality-bounded polymorphism, and general bounded polymorphism

- default implementations are a nice feature, too

Type classes take over:

- big part of standard library
- big part of reference manual
- temptation to go overboard with refining class hierarchy