Haskell

Many similarities with ML
• functions are first-class values
• strongly, statically typed
• polymorphic type system
• automatic type inference
• expression-oriented, recursion-oriented
• garbage-collected heap
• pattern matching
• highly regular and expressive

Key differences:
• lazy evaluation instead of eager evaluation
  • purely side-effect-free
  • modads for controlled side-effects, I/O, etc.
• type classes for more flexible polymorphic typechecking
  • simpler module system
• some interesting syntactic clean-ups and conveniences

Main design completed in 1992, by a committee, to unify many earlier lazy functional languages
• most recent version: Haskell 98

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 list = 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 (decls vs. exprs & output depends on implementation):
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] :: [Integer]

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

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
<fn> ::
  Tree Integer -> Tree Integer -> Tree Integer

size (Node 4 (Node 3 Empty Empty) Empty)
2 :: Integer

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

Uniform use of curried functions, including infix operators and data constructors

Type constructors use prefix notation, just like other functions

Layout & indentation are significant, and imply grouping and nesting
• can use { ... } to explicitly control grouping
**Sections**

Can call an infix operator on 0 or 1 of its arguments to create a curried function that takes the remaining argument(s)

\[ 3 + 4 \]
\[ ? :: \text{Integer} \]

\[ (+) \]
\[ <fn> :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ (+ 1) \quad \text{-- the increment function} \]
\[ <fn> :: \text{Integer} \rightarrow \text{Integer} \]

\[ (1 /) \quad \text{-- the inverse function} \]
\[ <fn> :: \text{Double} \rightarrow \text{Double} \]

Parentheses convert an infix operator into a prefix fn expression

Can treat a prefix fn name as an infix operator by bracketing with backquotes

\[ 6 \ ' \text{div} \ ' 2 \]
\[ 3 :: \text{Integer} \]

**List comprehensions**

Nice syntax for constructing a list from generators and guards:

\[ [ \ \text{expr} \mid \text{var} <- \text{expr}, \ldots, \text{boolExpr}, \ldots ] \]

\[ [ f \ x \mid x <- xs ] \quad \text{-- map } f \ x s \]
\[ [ (x,y) \mid x <- xs, y <- ys ] \quad \text{-- zip } xs \ ys \]
\[ [ y \mid y <- ys, y > 10 ] \quad \text{-- filter } (> 10) \ ys \]

\[ \text{quicksort } [] = [] \]
\[ \text{quicksort } (x:xs) = \text{quicksort } [y \mid y<x, y<-xs] \]
\[ \quad \quad \text{++ } [x] \]
\[ \quad \quad \text{++ quicksort } [y \mid y<-xs, y=x] \]

Arithmetic sequences easy to construct, too

\[ [1..10] \rightarrow [1,2,3,4,5,6,7,8,9,10] \]
\[ [2,4..10] \rightarrow [2,4,6,8,10] \]
\[ [2,4..] \rightarrow [2,4,6,8,10,12,...] \]
\[ [1..] \rightarrow [1,2,3,4,5,6,7,...] \]

**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’s value needed?

- when it’s being called as a function
- when it’s being used as the test of an if
- when it’s an operand of + (or some other primitive that can’t compute its result without looking at the value of its argument)
- when it’s being pattern-matched against (but then only enough to get the constructor tag; the components don’t need to be evaluated until they’re needed)
- if it’s the final result of the program

When is an expression not needed?

- when it’s not used
- when it’s just bound to another variable, e.g. a formal
- when it’s an argument of a data constructor

**Example**

\[ \text{my\_if } \text{test } \text{then\_val} \text{ else\_val } = \]
\[ \text{if } \text{test } \text{then } \text{then\_val} \text{ else } \text{else\_val} \]

\[ \text{my\_if True 3 4 } \rightarrow 3 \]
\[ \text{my\_if False 3 4 } \rightarrow 4 \]

\[ x = 3 \]
\[ y = 12 \]
\[ \text{my\_if } (x /= 0) \ (y \ ' \text{div} \ ' x) (-1) \rightarrow 4 \]
\[ \quad \text{-- different than in ML or Scheme!} \]

A call to \text{my\_if} doesn’t evaluate its arguments first
The test is always evaluated, since it’s needed to progress
Either the \text{then\_val} or the \text{else\_val} is evaluated, but not both

Needed “special form” in Scheme & ML to achieve this
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
  - “infinite” data structures,
    of which only a finite amount is ever actually used

Can always replace variable with defined expression
⇒ better equational reasoning

Evaluation order depends on what caller of function demands
⇒ hard to determine
- disallow side-effects, I/O, exceptions, etc.
  in (lazy) expressions
- use monads at outer level to get effects, in a specific order

Streams

Lists can be viewed as (possibly infinite) streams of values
- head, tail fields of a list structure won’t be evaluated
  until & unless they’re demanded

Lazy evaluation holds for all data structures in same way
- an infinite list of ascending integers, starting with n:
  `ints_from n = n : ints_from (n + 1)`
  [shorthand: `[n..]`

- the natural numbers:
  `nats = ints_from 0`  [shorthand: `[0..]`

- the perfect squares:
  `squares = map (^ 2) nats`
  → `[0, 1, 4, 9, 16, 25, ...`

- the fibonacci numbers:
  `fibs = 0 : 1 :
    [ a+b | (a,b) <- zip fibs (tail fibs)]`
  → `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...`

Simulating streams using first-class functions

Can simulate streams by wrapping lazy part(s) in function(s)

E.g. a lazy list: pair of functions to produce the head and the tail
on demand

```plaintext
- `datatype 'a lazy_list =
  lazy_nil |
  lazy_cons of (unit -> 'a) *
  (unit -> 'a lazy_list);

- `fun lazy_hd(lazy_cons(fh, _)) = fh();
val lazy_hd = fn : 'a lazy_list -> 'a

- `fun lazy_tl(lazy_cons(_, ft)) = ft();
val lazy_tl = fn : 'a lazy_list -> 'a lazy_list

- `fun first(_, _) = []
  = | first(n, lazy_cons(fh, ft)) =
    = fh() :: first(n-1, ft());
val first = fn : int * 'a lazy_list -> 'a list
```

A client

```plaintext
- `fun ints_from n =
  lazy_cons(fn()=n, fn()=ints_from(n+1));
val ints_from = fn : int -> int lazy_list

- `val nats = ints_from 0;
val nats = lazy_cons(fn, fn) : int lazy_list

- `val single_digits = first(10, nats);
[0,1,...,9] : int list
```

Will re-evaluate body of function each time head/tail of a
particular lazy list is referenced, unlike real lazy evaluation
- Scheme builds in delay and force to avoid this

Have to have multiple versions of list operations like `map, fold, etc.,` for eager vs. lazy lists, unlike real lazy evaluation
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, e.g.
  mapping & filtering & combining & (un)zipping streams
• scanner produces a stream of tokens
• input produces a stream of characters
• event-driven simulations produce streams of events
• ....

Don’t worry about controlling how much to generate; generate everything that might possibly be useful

Independently produce operations to manipulate and extract interesting subset of generated data
• only portion needed in final result will actually be generated

Example

Implement scanner as a generator of a stream of tokens

Implement utility that checks which functions have been changed since last compile
• generate streams of tokens on both versions
• compares two streams to find difference
• if difference found, rest of tokens won’t be demanded, therefore won’t be generated

Implement parser to produce a stream of possible parses, if grammar has type-dependent ambiguities (like C++)
• consumes stream of tokens, until first syntax error

Implement typechecker to consume possible parse trees, filter for those that typecheck

I/O

How can a purely functional program interact with the outside world, e.g. read any (mutable) input or produce any output?

Idea:
• introduce a special IO type, whose values are I/O actions that could be performed
• top-level main function yields an I/O action, which is performed only “when main returns”
• but lazy evaluation makes this happen “as soon as possible”

IO data type is a special case of a monad
• very powerful mechanism for controlling & encapsulating effects of many sorts, including mutable state, exceptions, resource consumption, etc.

IO actions

IO a: the type of actions that have some I/O effect and then yield a value of type a

main :: IO ()
• main returns an I/O action that has no result
  • the system runs a program by demanding the result of main, and executing the actions that are computed

Some basic I/O actions:
• getChar :: IO Char
• putChar :: Char -> IO ()
• openFile :: String -> IOMode -> IO Handle
• hClose :: Handle -> IO ()
• stdin, stdout :: IO Handle
• hGetChar :: Handle -> IO Char
• hPutChar :: Handle -> Char -> IO ()
• hGetContents :: Handle -> IO String

A no-op action:
  return expr :: IO typeOfExpr
  • does no I/O but yields a value
Composite actions

Can combine actions together, in sequences:

\[
\text{do } v_1 \leftarrow \text{action}_1 \\
v_2 \leftarrow \text{action}_2 \\
\ldots \\
\text{action}_n
\]

- yields an action that, if performed, first performs action\(_1\), binding the result value to \(v_1\), then performs action\(_2\), binding the result value to \(v_2\), ..., then performs action\(_n\) and returns its result value
- any of the \(v_i\) are optional

Example: a program that copies its input to its output, twice

\[
\begin{align*}
\text{hPutString} & : \text{Handle} \rightarrow \text{String} \rightarrow \text{IO} () \\
\text{hPutString} h [] & = \text{return} () \\
\text{hPutString} h (x:xs) & = \text{do } \text{hPutChar} h x \\
& \quad \text{hPutString} h xs \\
\text{main} & : \text{IO} () \\
\text{main} & = \text{do } \text{contents} \leftarrow \text{hGetContents stdin} \\
& \quad \text{hPutString} \text{stdout contents} \\
& \quad \text{hPutString} \text{stdout contents}
\end{align*}
\]

The magic

Key property of the IO data type:
- there are no functions to perform an action, yielding something without IO in its result type
  - the only way to perform an action is to have main return (an action containing) it

Corollary: can't embed I/O (or any other kind of side-effect) in an expression that doesn't yield an I/O action!

Type structure enforces a strict separation between purely effect-free computations (result type \(\equiv \text{IO a}\)) and (potentially) effect-full computations (result type \(\equiv \text{IO a}\))
- effect-full computations are at the “top level” of the computation
  - effect-free computations are its subexpressions
- effect-full computations are explicitly sequenced using do

Effects and lazy evaluation

Lazy evaluation doesn’t interact badly with effects, since none of the effects are actually performed until main returns
- but nothing is computed until it’s demanded...

Operation of a Haskell program:
- Haskell runtime system demands result I/O action of main be computed and performed
- This demands evaluation & performance of e.g. a do block action
- This demands evaluation & performance of the first action in the do block
- Etc., until some primitive action is reached, at which point Haskell’s runtime system performs it, and then proceeds to the next action subexpression

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.

\[
\text{fun square } n = n * n;
\]

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 overleadings

With the one oddball exception of equality types, ML supports only unbounded parametric polymorphism
Bounded polymorphism

Would like to allow **bounded** polymorphism, constraining possible instantiating types in order to be able to call specialized operations on them

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 3 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`
  - `method` `+` : `(number)\rightarrow number;`
  - `method` `*` : `(number)\rightarrow number;`
  - ...
- `class` `int` `subtypes` `number` { ...
- `class` `float` `subtypes` `number` { ...

```
- `fun` `square` `n` = `n` \* `n;
val` `square` = `fn` : `number` \rightarrow `number;
- `square` `3;
9` : `number`
- `square` `3.4;
11.5` : `number`
```

[How to get result type to be as precise 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
- global names \(\Rightarrow\) cannot overload with other global names
- `a` is a placeholder name for 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`
  `x` `==` `y` = `intEq` `x` `y`
  `x` `/=` `y` = `intNeq` `x` `y`
```

```
instance` `Eq` `Float` `where`
  `x` `==` `y` = `floatEq` `x` `y`
  `x` `/=` `y` = `floatNeq` `x` `y`
```

Now can invoke type class operations on member types:

```
3` `==` `4` -- allowed; calls `intEq`
3.4` `/=` `5.6` -- allowed; calls `floatNeq`
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 \langle x_1, y_1 \rangle \ (x_2, y_2) = x_1 = = x_2 \ & \ & y_1 = = y_2
```

```
eq_pair :: (Eq a, Eq b) \Rightarrow \ (a, b) \rightarrow \ (a, b) \rightarrow \text{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 \rightarrow \ a \rightarrow \ \ [a] \rightarrow \ \text{Bool}
member \ [\] \ = \ False
member x \ (y:ys) = \ x = = y \ || \ 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) \rightarrow \ Eq \ (a, b) \ where
\ (x_1, y_1) = = \ (x_2, y_2) = x_1 = = x_2 \ & \ & y_1 = = y_2
x = /= y = = \ not \ (x = = y)
```

“A list of a supports \-- if a does“

```
instance Eq a \rightarrow \ Eq \ [a] \ where
[\] = = \ [\] = \ True
(x:xs) = = \ (y:ys) = \ x = = y \ & \ & xs = = ys
_ = = _ = \ False
x = /= y = = \ not \ (x = = y)
```

Default implementations in type classes

Add a /= operation, which defaults to negating

```
class Eq a where
\ (\=), (\/=) :: \ a \rightarrow \ a \rightarrow \text{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) \rightarrow \ Eq \ (a, b) \ where
\ (x_1, y_1) = = \ (x_2, y_2) = x_1 = = x_2 \ & \ & y_1 = = y_2
-- inherit default /=
```

Type subclasses

Can define new type classes that

- extend existing type classes & add new operations
- define the superclass(es) as contexts
- for a type to be an instance of a subclass, it must already be an instance of all its superclasses
- multiple inheritance allowed
- name clashes can’t happen since operations not overloading

Example: Ord class of totally ordered things, subclassing Eq

```
class Eq a \rightarrow \ Ord a \ where
-- Ord “inherits“ Eq operations \& = =
\ (<), (\<=), (\>=), (\>) :: \ a \rightarrow \ a \rightarrow \text{Bool}
min, max :: \ a \rightarrow \ a \rightarrow \ a
x \ <= \ y = = x \ = = y \ or \ x \ < \ y
min x y = = \ if \ x \ < \ y \ then \ x \ else \ y
\ ...
\ (\>=, \ & \ & \ max \ defaulted \ too) \ ...
```

A client function:

```
member_sorted :: Ord a \rightarrow \ a \rightarrow \ [a] \rightarrow \text{Bool}
member_sorted \ [\] \ = \ False
member_sorted x \ (y:ys) = =
x = = y \ || \ x < y \ & \ & \ member_sorted x ys
```

Another example:

```
member :: Eq a \rightarrow \ a \rightarrow \ \ [a] \rightarrow \ \text{Bool}
member \ [\] \ = \ False
member x \ (y:ys) = =
x = = y \ || \ x < y \ & \ & \ member x ys
```

```
member_sorted :: Ord a \rightarrow \ a \rightarrow \ [a] \rightarrow \text{Bool}
member_sorted \ [\] \ = \ False
member_sorted x \ (y:ys) = =
x = = y \ || \ x < y \ & \ & \ member_sorted x ys
```

```
member Sorted :: Ord a \rightarrow \ a \rightarrow \ [a] \rightarrow \text{Bool}
member Sorted \ [\] \ = \ False
member Sorted x \ (y:ys) = =
x = = y \ || \ x < y \ & \ & \ member_sorted x ys
```
Ord instances

(assume Eq instances already declared)

```haskell
instance Ord Int where
  x < y = intLt x y
  x <= y = intLeq x y
  ... -- other operations implemented or inherited
```

```haskell
instance (Ord a, Ord b) => Ord (a,b) where
  (x1,y1) < (x2,y2) = x1 < x2 || x1==x2 && y1 < y2
  -- all other operations inherited
```

```haskell
instance Ord a => Ord [a] where
  []    < (y:ys) = True
  (x:xs) < (y:ys) = x < y || x==y && xs < ys
  _    < _    = False
  -- all other operations inherited
```

Hierarchy of some predefined type classes

```
Ord
  Eq
  Show
  Read

(a)

Ix
(t)

Enum
Num

Bool
Char

Integral
Real
Fractional

Int
Integer

RealFrac

Floating

RealFloat

Float
Double
```

Type classes vs. ML polymorphism

ML polymorphism is simple, but has warts:
- "equality-bounded" polymorphism
- overloaded operators, not polymorphism

Haskell's type classes subsume and unify
unbounded polymorphism, equality-bounded polymorphism,
and general bounded polymorphism
- default implementations are a nice feature, too

But type classes take over the language
- big part of standard library
- big part of reference manual
- temptation to go overboard with refining class hierarchy
- [just like OO languages]

Type classes vs. OO classes

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)
- [Haskell extensions with existential types can do this]

No inheritance, other than single default method

Type classes support binary operations like -- and + well,
where the arguments and result are all of same type

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

- hard to do in an OO language without
  F-bounded subtype polymorphism or similar feature

Retain type inference, unlike OO languages