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 {
    method +:(number)→number;
    method *:(number)→number;
    ...
  };
- class int subtypes number { ... };
- class float subtypes number { ... };

- fun square n = n * n;
val square = fn : number → 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
- `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) ...
```

A client function:

```
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

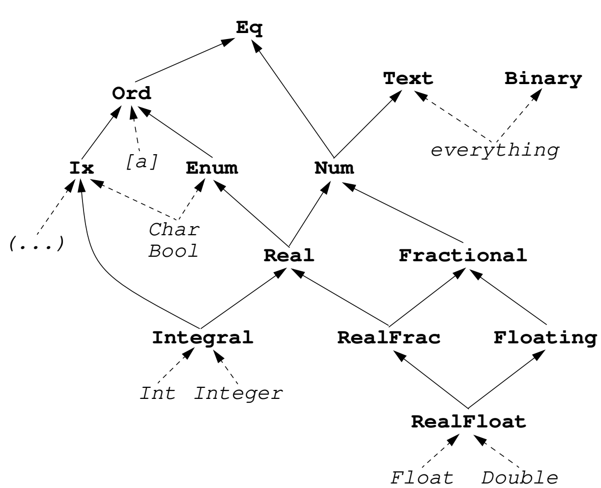
(assume `Eq` instances already declared)

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

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

```
instance (Ord a)=> 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)
- 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 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 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

Type classes take over:

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