Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 22

- Functional programming languages
- Introduction to Hakell
- Type Classes
- (Type) Constructor Classes

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  - different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics
- These results led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well
  - this conjecture is known as *Church's thesis*

Historical Origins

- Church's model of computing is called the *lambda calculus*
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter *λ,* hence the notation's name)
  - allows one to define mathematical functions in a constructive/effective way
  - Lambda calculus was the inspiration for functional programming
  - computation proceeds by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Functional Programming Concepts

- Functional languages such as **Lisp**, **Scheme**, **FP**, **ML**, **Miranda**, and **Haskell** are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
  - no mutable state
  - no side effects

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - recursion
    - Takes the place of iteration
  - powerful list facilities
    - Recursive function exploit recursive definition of lists
  - serious polymorphism
    - Relevance of Container/Collections
  - fully general aggregates
    - Data structures cannot be modified, have to be re-created
  - structured function returns
  - garbage collection
    - Unlimited extent for locally allocated data structures

Other Related Concepts

- **Lisp** also has some features that are not necessary present in other functional languages:
  - programs are data
  - self-definition
  - read-evaluate-print interactive loop
- Variants of LISP
  - (Original) Lisp: purely functional, dynamically scoped as early variants
  - Common Lisp: current standard, statically scoped, very complex
  - Scheme: statically scoped, very elegant, used for teaching

Other functional languages: the ML family

- Robin Milner (Turing award in 1991, CCS, Pi-calculus, …)
- Statically typed, general-purpose programming language
  - "Meta-Language" of the LCF theorem proving system
- Type safe, with type inference and formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions

Other functional languages: Haskell

- Designed by committee in 80's and 90's to unify research efforts in lazy languages
  - Evolution of Miranda
  - Haskell 1.0 in 1990, Haskell '98, Haskell' ongoing
- Several features in common with ML, but **some differ**:
- Types and type checking
  - Type inference
  - Parametric polymorphism
  - **Ad hoc polymorphism (aka overloading)**
- Control
  - **Lazy vs. eager evaluation**
  - Tail recursion and continuations
- Purely functional
  - **Precise management of effects**
  - Rise of multi-core, parallel programming likely to make minimizing state much more important

The Glasgow Haskell Compiler [GHC]
www.haskell.org/platform

Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions

Overview of Haskell

- Interactive Interpreter (ghci): read-eval-print
  - ghci infers type before compiling or executing
  - Type system does not allow casts or similar things!
- Examples

```
Prelude> (5+3)-2
6
it :: Integer
Prelude> if 5>3 then "Harry" else "Hermione"
"Harry"
it :: [Char]       -- String is equivalent to [Char]
Prelude> 5==4
False
it :: Bool
```

Overview by Type

- Booleans

```
True, False :: Bool
if …  then … else …    --types must match
```

- Integers

```
0, 1, 2, … :: Integer
+, * , …   :: Integer  -> Integer -> Integer
```

- Strings

```
"Ron Weasley"
```

- Floats

```
1.0, 2, 3.14159, …  --type classes to disambiguate
```

Simple Compound Types

- Tuples

```
(4, 5, "PLP") :: (Integer, Integer, String)
```

- Lists

```
[] :: [a]                           -- NIL, polymorphic type
1 : [2, 3, 4] :: [Integer]    -- infix cons notation
[1,2]++[3,4] :: [Integer]    -- concatenation
```

- Records

```
data Person = Person {firstName :: String,
                      lastName  :: String}
hg = Person { firstName = “Hermione”,
              lastName  = “Granger”}
```

More on list constructors

```
ghci> [1..20]          -- ranges
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> [3,6..20]        -- ranges with step
[3,6,9,12,15,18]
ghci> [7,6..1]
[7,6,5,4,3,2,1]
```

```
ghci> take 10 [1..]          -- (prefix of) infinite lists
[1,2,3,4,5,6,7,8,9,10]
ghci> take 10 (cycle [1,2])
[1,2,1,2,1,2,1,2,1,2]
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

How does it work???

Laziness

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them

```
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

- Programmers can write control-flow operators that have to be built-in in eager languages

Short-circuiting "or"

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

Applicative and Normal Order evaluation

- Applicative Order evaluation
  - Arguments are evaluated before applying the function – aka *Eager evaluation, parameter passing by value*
- Normal Order evaluation
  - Function evaluated first, arguments if and when needed
  - Sort of *parameter passing by name*
  - Some evaluation can be repeated
- Church-Rosser
  - If evaluation terminates, the result (*normal form*) is unique
  - If some evaluation terminates, normal order evaluation terminates

Define **Ω = (λx.x x)**

Then
ΩΩ = (λx.x x) (λx.x x)
→ x x [(λx.x x)/x]
→ (λx.x x) (λx.x x) = ΩΩ
→ … *non-terminating*

(λx. 0) (ΩΩ)
→ { *Applicative order*}
… *non-terminating*

(λx. 0) (ΩΩ)
→ { *Normal order*}
0

β-conversion
(λx.t) t' = t [t'/x]

Applicative order
(λx.(+ x x)) (+ 3 2)
→ (λx.(+ x x)) 5
→ (+ 5 5)
→ 10

Normal order
(λx.(+ x x)) (+ 3 2)
→ (+ (+ 3 2) (+ 3 2))
→ (+ 5 (+ 3 2))
→ (+ 5 5)
→ 10

Relating evaluation order to Parameter Passing Mechanisms

- Parameter passing modes
  - In
  - In/out
  - Out
- Parameter passing mechanisms
  - Call by value (in)
  - Call by reference (in+out)
  - Call by result (out)
  - Call by value/result (in+out)
  - Call by name (in+out)
- Different mechanisms used by C, Fortran, Pascal, C++, Java, Ada (and Algol 60)

Call by name & Lazy evaluation (*call by need*)

- In ***call by name*** parameter passing (default in Algol 60) arguments (like expressions) are passed as a closure ("thunk") to the subroutine
- The argument is (re)evaluated each time it is used in the body
- Haskell realizes ***lazy evaluation*** by using ***call by need*** parameter passing, which is similar: an expression passed as argument is evaluated only if its value is needed.
- Unlike ***call by name,*** the argument is evaluated *only the first time*, using ***memoization***: the result is saved and further uses of the argument do not need to re-evaluate it
- Combined with *lazy data constructors*, this allows to construct potentially infinite data structures and call infinitely recursive functions without necessarily causing non-termination
- Lazy evaluation works fine with **purely functional** languages
- Side effects require that the programmer reason about the order that things happen, not predictable in lazy languages.

Patterns and Declarations

- Patterns can be used in place of variables

  <pat> ::= <var> | <tuple> | <cons> | <record> ...

- Value declarations
  - General form: <pat> = <exp>
  - Examples

```
myTuple = ("Foo", "Bar")
(x,y)  = myTuple   -- x = "Foo", y = "Bar"
myList = [1, 2, 3, 4]
z:zs   = myList     -- z = 1, zs = [2,3,4]
```

  - Local declarations

```
let (x,y) = (2, "FooBar") in x * 4
```

Functions and Pattern Matching

- Anonymous function

```
\x -> x+1      --like Lisp lambda, function (…) in JS
```

- Function declaration form

  <name> <pat$_1$> = <exp$_1$>

  <name> <pat$_2$> = <exp$_2$> …

  <name> <pat$_n$> = <exp$_n$> …

- Examples

```
f (x,y) = x+y    --argument must match pattern (x,y)

length [] = 0
length (x:s) = 1 + length(s)
```

More Functions on Lists

- Apply function to every element of list

```
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]                    [2,3,4]
```

- Reverse a list

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

```
reverse xs =
    let rev ( [], accum ) = accum
        rev ( y:ys, accum ) = rev ( ys, y:accum )
    in rev ( xs, [] )
```

List Comprehensions

- Notation for constructing new lists from old:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to "set comprehension"

$$\{ x \mid x \in Odd \;\wedge\; x > 6 \}$$

More on List Comprehensions

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20] -- more predicates

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]    -- more lists

length xs = sum [1 | _ <- xs] -- anonymous (don't care) var

-- strings are lists…
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Datatype Declarations

- Examples

```
data Color = Red | Yellow | Blue
```

elements are Red, Yellow, Blue

```
data Atom = Atom String | Number Int
```

elements are Atom “A”, Atom “B”, …, Number 0, …

```
data List    = Nil   |   Cons (Atom, List)
```

elements are Nil, Cons(Atom “A”, Nil), …

Cons(Number 2, Cons(Atom(“Bill”), Nil)), ...

- General form

```
data <name> = <clause> | … | <clause>
<clause> ::= <constructor> | <contructor> <type>
```

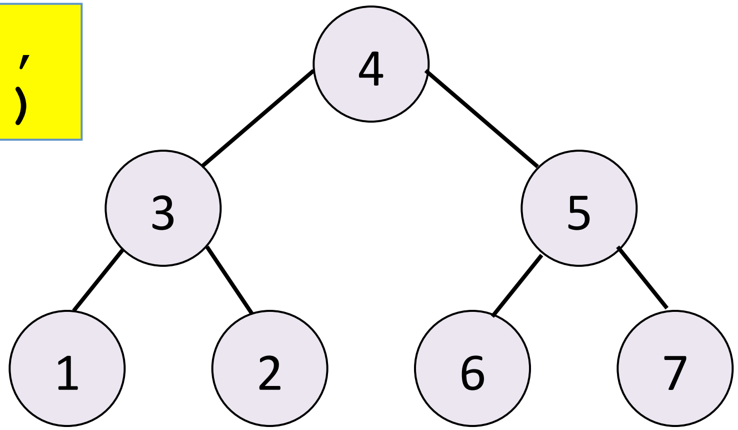
– Type name and constructors must be Capitalized.

Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node(4, Node(3, Leaf 1, Leaf 2),
        Node(5, Leaf 6, Leaf 7))
```



- Constructors can be used in Pattern Matching
- Recursive function

```
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Case Expression

- Datatype

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```
case e of
     Var n ->  …
     Const n -> …
     Plus(e1,e2) -> …
```

Indentation matters in case statements in Haskell.

Function Types in Haskell

In Haskell, **`f :: A -> B`** means for every x $\in$ A,

$$f(x) = \begin{cases} \text{some element } y = f(x) \in B \\ \text{run forever} \end{cases}$$

In words, "if f(x) terminates, then f(x) $\in$ B."

In ML, functions with type A $\to$ B can throw an exception or have other effects, but not in Haskell

```
ghci> :t not    -- type of some predefined functions
not :: Bool -> Bool
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t not
not :: Bool -> Bool
ghci> :t (:)
(:) :: a -> [a] -> [a]
ghci> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Note: if *f* is a standard binary function, *\`f\`* is its infix version
If *x* is an infix (binary) operator, *(x)* is its prefix version.

Higher-Order Functions

- Functions that take other functions as arguments or return as a result are higher-order functions.
- Common Examples:
  - **Map**: applies argument function to each element in a collection.
  - **Reduce**: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> let list = [1,2,3]
ghci> map (\x -> x+1) list
[2,3,4]
ghci> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
ghci> foldl (\accum i -> i + accum) 0 list
6
```

Searching a substring: Java code

```
static int indexOf(char[] source, int sourceOffset, int sourceCount,
                   char[] target, int targetOffset, int targetCount,
                   int fromIndex) {
    ...

    char first  = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        /* Look for first character. */
        if (source[i] != first) {
            while (++i <= max && source[i] != first);
        }

        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
            int j = i + 1;
            int end = j + targetCount - 1;
            for (int k = targetOffset + 1; j < end && source[j] ==
                     target[k]; j++, k++);

            if (j == end) {
                /* Found whole string. */
                return i - sourceOffset;
    }   }   }
    return -1;
  }
```

Searching a Substring: Exploiting Laziness

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
-- returns True if first list is prefix of the second
isPrefixOf [] x = True
isPrefixOf (y:ys) [] = False
isPrefixOf (y:ys)(x:xs) =
  if (x == y) then isPrefixOf ys xs else False
```

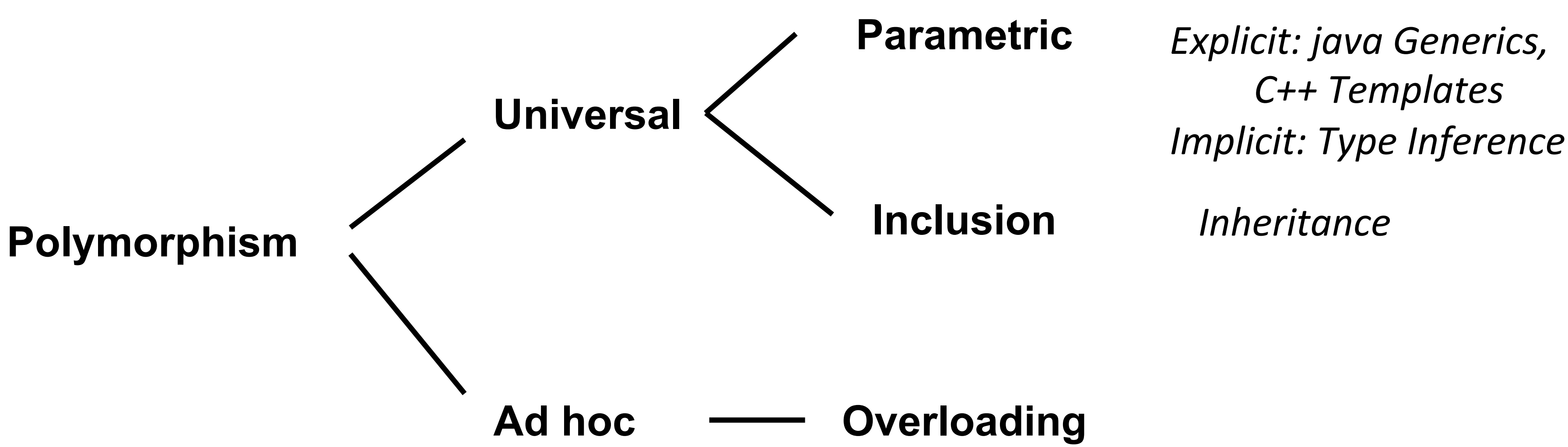
```
suffixes:: String -> [String]
-- All suffixes of s
suffixes[]      = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs
```

```
or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []     = False
or (b:bs) = b || or bs
```

```
isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t
                       | t <- suffixes s ]
```

Polymorphism

- The ability of associating a single interface with entities of different types
- We focus on *polymorphic functions,* applicable to arguments of different types



Generic Polymorphism in Haskell

- Type declarations not necessary: ***Type Inference*** algorithm computes the most general type of a declared function
- Based on the original algorithm invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958
- Revised and extended by Hindley (1969) and Milner (1978)
- Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C++0x,…
- Guaranteed to produce the ***most general type***.
- Example of a flow-insensitive static analysis algorithm
- You will study it in next semester…

Polymorphism vs Overloading

- (Parametric) polymorphism
  - ***Single*** algorithm may be given many types
  - Type variable (implicit or explicit) may be replaced by any type (almost... -> *bounded polymorphism*)
  - if `f::t→t` then `f::Int→Int`, `f::Bool→Bool`, ...
- Overloading
  - A single symbol may refer to more than one algorithm.
  - Each algorithm may have different type.
  - Choice of algorithm determined by type context.
  - `+` has types `Int → Int → Int` and `Float → Float → Float`, but not `t→t→t` for arbitrary `t`.

Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions

Why Overloading?

- Many useful functions are not parametric
- Can list membership work for any type?

```
member :: [w] -> w -> Bool
```

  - No! Only for types w for that support equality.
- Can list sorting work for any type?

```
sort :: [w] -> [w]
```

  - No! Only for types w that support ordering.

Overloading Arithmetic, Take 1

- Allow functions containing overloaded symbols to define multiple functions:

```
square x = x * x           -- legal
-- Defines two versions:
-- Int -> Int and Float -> Float
```

- But consider:

```
squares (x,y,z) =
    (square x, square y, square z)
-- There are 8 possible versions!
```

- Approach not widely used because of exponential growth in number of versions.

Overloading Arithmetic, Take 2

- Basic operations such as + and * can be overloaded, but not functions defined from them

```
3 * 3                 -- legal
3.14 * 3.14           -- legal
square x = x * x      -- Int -> Int
square 3              -- legal
square 3.14           -- illegal
```

- Standard ML uses this approach.
- Not satisfactory: Programmer cannot define functions that implementation might support

Overloading Equality, Take 1

- Equality defined only for types that admit equality: types not containing **function** or **abstract types**.

```
3 * 3 == 9               -- legal
'a' == 'b'               -- legal
\x->x == \y->y+1         -- illegal
```

- Overload equality like arithmetic ops + and * in SML.
- But then we can't define functions using '==':

```
member [] y      = False
member (x:xs) y = (x==y) || member xs y

member [1,2,3] 3         -- ok if default is Int
member "Haskell" 'k'     -- illegal
```

- Approach adopted in first version of SML.

Overloading Equality, Take 2

- Make type of equality fully polymorphic

```
(==) :: a -> a -> Bool
```

- Type of list membership function

```
member :: [a] -> a -> Bool
```

- **Miranda** used this approach.
  - Equality applied to a **function** yields a runtime error
  - Equality applied to an **abstract type** compares the underlying representation, which violates abstraction principles

Overloading Equality, Take 3

- Make equality polymorphic **in a limited way**:

```
(==) :: a(==) -> a(==) -> Bool
```

where a(==) is type variable restricted to types with equality

- Now we can type the member function:

```
member :: a(==) -> [a(==)] -> Bool
member  4           [2,3] :: Bool
member 'c'          ['a', 'b', 'c'] :: Bool
member (\y->y *2) [\x->x, \x->x + 2]  -- type error
```

- Approach used in SML today, where the type a(==) is called an "eqtype variable" and is written ''a.

Type Classes

- Type classes solve these problems
  - Provide concise types to describe overloaded functions, so no exponential blow-up
  - Allow users to define functions using overloaded operations, eg, square, squares, and member
  - Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged built-ins
  - Generalize ML's eqtypes to arbitrary types
  - Fit within type inference framework

Intuition

- A function to sort lists can be passed a comparison operator as an argument:

```
qsort:: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
              ++ [x] ++
                  qsort cmp (filter (not.cmp x) xs)
```

  – This allows the function to be parametric

- We can built on this idea …

Intuition (continued)

- Consider the "overloaded" parabola function

```
parabola x = (x * x) + x
```

- We can rewrite the function to take the operators it contains as an argument

```
parabola' (plus, times) x = plus (times x x) x
```

  – The extra parameter is a "dictionary" that provides implementations for the overloaded ops.
  – We have to rewrite all calls to pass appropriate implementations for plus and times:

```
y = parabola'(intPlus,intTimes) 10
z = parabola'(floatPlus, floatTimes) 3.14
```

Systematic programming style

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p

get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t

-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus  = get_plus  dict
                      times = get_times dict
                  in plus (times x x) x
```

Type class declarations will generate Dictionary type and selector functions

Systematic programming style

Type class **instance declarations** produce instances of the Dictionary

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Dictionary construction
intDict   = MkMathDict intPlus   intTimes
floatDict = MkMathDict floatPlus floatTimes

-- Passing dictionaries
y = parabola intDict   10
z = parabola floatDict 3.14
```

Compiler will add a dictionary parameter and rewrite the body as necessary

Type Class Design Overview

- Type class declarations
  - Define a set of operations, give the set a name
  - Example: `Eq a` type class
    - operations `==` and `\=` with `type a -> a -> Bool`
- Type class instance declarations
  - Specify the implementations for a particular type
  - For `Int` instance, `==` is defined to be integer equality
- Qualified types (or Type Constraints)
  - Concisely express the operations required on otherwise polymorphic type

```
member:: Eq w => w -> [w] -> Bool
```

Qualified Types

"for all types w that support the Eq operations"

```
Member :: Eq w => w -> [w] -> Bool
```

If a function works for every type with particular properties, the type of the function says just that:

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a -> String
square    :: Num n  => n -> n
squares   ::(Num t, Num t1, Num t2) =>
                 (t, t1, t2) -> (t, t1, t2)
```

Otherwise, it must work for any type

```
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

Type Classes

Works for any type 'n' that supports the Num operations

```
square :: Num n => n -> n
square x = x*x
```

The class declaration says what the Num operations are

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
  ...etc...
```

An instance declaration for a type T says how the Num operations are implemented on T's

```
instance Num Int where
  a + b    = intPlus  a b
  a * b    = intTimes a b
  negate a = intNeg a
  ...etc...
```

```
intPlus  :: Int -> Int -> Int
intTimes :: Int -> Int -> Int
etc, defined as primitives
```

Compiling Overloaded Functions

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

The "Num n =>" turns into an extra value argument to the function. It is a value of data type Num n and it represents a dictionary of the required operations.

A value of type (Num n) is a dictionary of the Num operations for type n

Compiling Type Classes

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

```
class Num n where
  (+)    :: n -> n -> n
  (*)    :: n -> n -> n
  negate :: n -> n
  ...etc...
```

The class decl translates to:
A data type decl for Num
A selector function for each class operation

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
data Num n
  = MkNum (n -> n -> n)
          (n -> n -> n)
          (n -> n)
          ...etc...
...
(*) :: Num n -> n -> n -> n
(*) (MkNum _ m _ ...) = m
```

A value of type (Num n) is a dictionary of the Num operations for type n

Compiling Instance Declarations

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b    = intPlus  a b
  a * b    = intTimes a b
  negate a = intNeg a
  ...etc...
```

```
dNumInt :: Num Int
dNumInt = MkNum intPlus
                intTimes
                intNeg
                ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num n) is a dictionary of the Num operations for type n

Implementation Summary

- The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: **the dictionary**.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
- The compiler converts each type class declaration into a dictionary type declaration and a set of selector functions.
- The compiler converts each instance declaration into a dictionary of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a dictionary. It uses the static, qualified type of the function to select the dictionary.

Functions with Multiple Dictionaries

```
squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares(x,y,z) = (square x, square y, square z)
```

Note the concise type for the squares function!

```
squares :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
squares (da,db,dc) (x, y, z) =
                 (square da x, square db y, square dc z)
```

Pass appropriate dictionary on to each square function.

Compositionality

Overloaded functions can be defined from other overloaded functions:

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x)
                    (square d y)
```

Extract addition operation from d

Pass on d to square

Compositionality

Build compound instances from simpler ones:

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

instance Eq Int where
   (==) = intEq      -- intEq primitive equality

instance (Eq a, Eq b) => Eq(a,b)
   (u,v) == (x,y)      = (u == x) && (v == y)

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

Compound Translation

Build compound instances from simpler ones.

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

```
data Eq = MkEq (a->a->Bool)    -- Dictionary type
(==) (MkEq eq) = eq            -- Selector
dEqList :: Eq a -> Eq [a]      -- List Dictionary
dEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```

Many Type Classes

- **Eq**: equality
- **Ord**: comparison
- **Num**: numerical operations
- **Show**: convert to string
- **Read**: convert from string
- **Testable**, **Arbitrary**: testing.
- **Enum**: ops on sequentially ordered types
- **Bounded**: upper and lower values of a type
- Generic programming, reflection, monads, …
- And many more.

Subclasses

- We could treat the Eq and Num type classes separately

```
memsq :: (Eq a, Num a) => a -> [a] -> Bool
memsq x xs = member (square x) xs
```

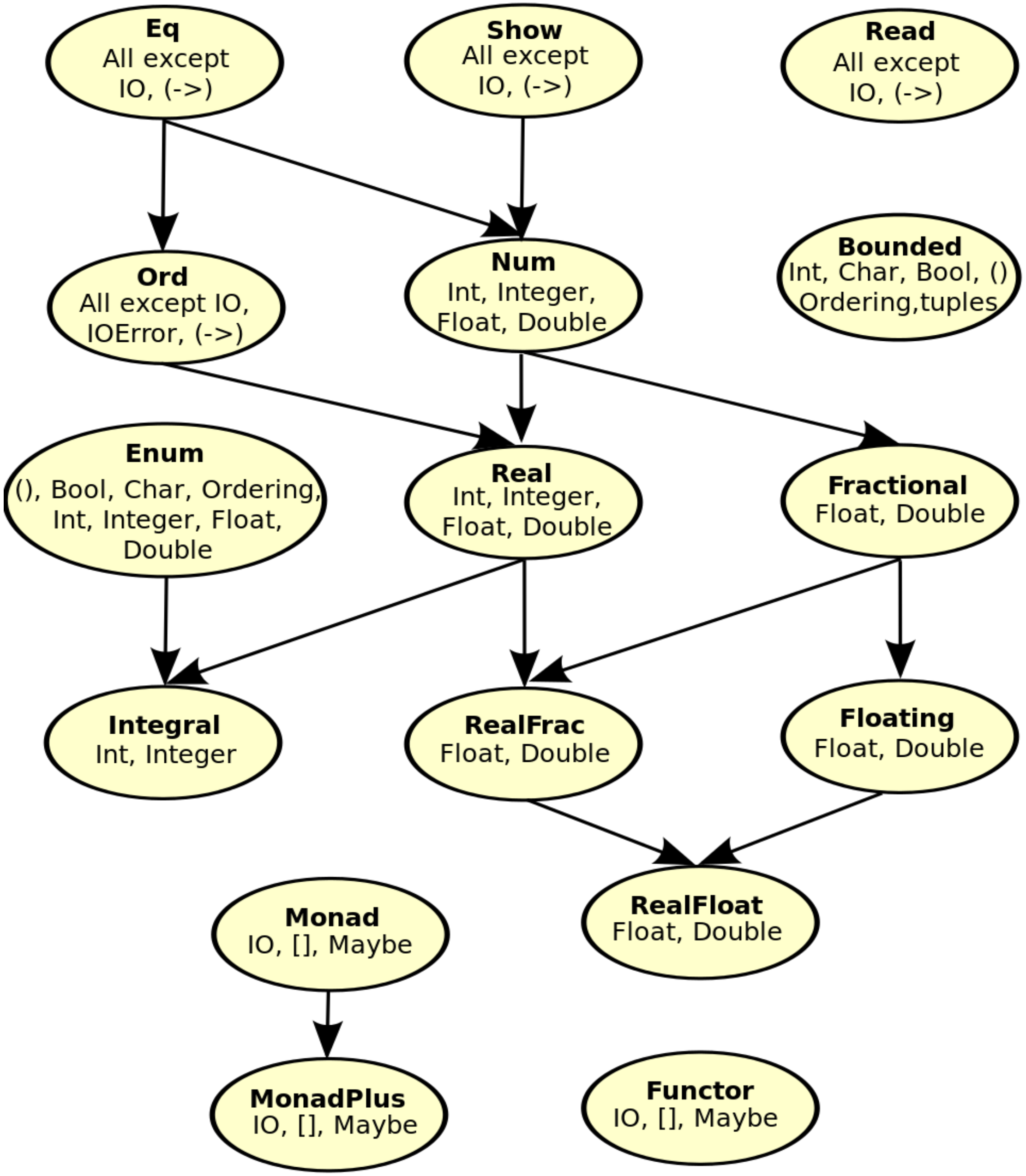
  – But we expect any type supporting Num to also support Eq

- A subclass declaration expresses this relationship:

```
class Eq a => Num a where
   (+) :: a -> a -> a
   (*) :: a -> a -> a
```

- With that declaration, we can simplify the type of the function

```
memsq :: Num a => a -> [a] -> Bool
memsq x xs = member (square x) xs
```


Eq
All except IO, (->)
Show
All except IO, (->)
Read
All except IO, (->)
Ord
All except IO, IOError, (->)
Num
Int, Integer, Float, Double
Bounded
Int, Char, Bool, ()
Ordering,tuples
Enum
(), Bool, Char, Ordering, Int, Integer, Float, Double
Real
Int, Integer, Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

Default Methods

- Type classes can define “default methods”

```
-- Minimal complete definition:
--      (==) or (/=)
class Eq a where
    (==) :: a -> a -> Bool
    x == y    =  not (x /= y)
    (/=) :: a -> a -> Bool
    x /= y    =  not (x == y)
```

- Instance declarations can override default by providing a more specific definition.

Deriving

- For Read, Show, Bounded, Enum, Eq, and Ord, the compiler can generate instance declarations automatically

```
data Color = Red | Green | Blue
     deriving (Show, Read, Eq, Ord)
```

```
Main> show Red
“Red”
Main> Red < Green
True
Main>let c :: Color = read “Red”
Main> c
Red
```

  – *Ad hoc* : derivations apply only to types where derivation code works

Numeric Literals

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ...

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded.
1 :: (Num a) => a

"1" means "fromInteger 1"

Advantages:
- Numeric literals can be interpreted as values of any appropriate numeric type
- Example: 1 can be an Integer or a Float or a user-defined numeric type.

Type Inference

- Type inference infers a qualified type Q => T
  - T is a Hindley Milner type, inferred as usual
  - Q is set of type class predicates, called a constraint
- Consider the example function:

```
example z xs =
   case xs of
     []      -> False
     (y:ys) -> y > z || (y==z && ys == [z])
```

  - Type T is a -> [a] -> Bool
  - Constraint Q is { Ord a, Eq a, Eq [a]}

Ord a	because	y>z
Eq a	because	y==z
Eq [a]	because	ys == [z]

Type Inference

- Constraint sets Q can be simplified:
  - Eliminate duplicates
    - {Eq a, Eq a} simplifies to {Eq a}
  - Use an **instance declaration**
    - If we have instance Eq a => Eq [a],
    - then {Eq a, Eq [a]} simplifies to {Eq a}
  - Use a **subclass declaration**
    - If we have class Eq a => Ord a where ...,
    - then {Ord a, Eq a} simplifies to {Ord a}
- Applying these rules,
  - {Ord a, Eq a, Eq[a]} simplifies to {Ord a}

Type Inference

- Putting it all together:

```
example z xs =
   case xs of
     []      -> False
     (y:ys) -> y > z || (y==z && ys ==[z])
```

  - T = a -> [a] -> Bool
  - Q = {Ord a, Eq a, Eq [a]}
  - Q simplifies to {Ord a}
  - `example :: {Ord a} => a -> [a] -> Bool`

Detecting Errors

- Errors are detected when predicates are known not to hold:

```
Prelude> 'a' + 1
 No instance for (Num Char)
      arising from a use of `+' at <interactive>:1:0-6
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 'a' + 1
    In the definition of `it': it = 'a' + 1
```

```
Prelude> (\x -> x)
 No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>:1:0-4
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a stmt of a 'do' expression: print it
```

More Type Classes: Constructors

- **Type Classes** are predicates over **types**
- **[Type] Constructor Classes** are predicates over **type constructors**
- Example: Map function useful on many Haskell types
- Lists:

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

result = map (\x->x+1) [1,2,4]
```

Constructor Classes

- More examples of map function

```
data Tree a = Leaf a | Node(Tree a, Tree a)
    deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)

t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
result = mapTree (\x->x+1) t1
```

```
data Opt a = Some a | None
 deriving Show

mapOpt :: (a -> b) -> Opt a -> Opt b
mapOpt f None = None
mapOpt f (Some x) = Some (f x)

o1 = Some 10
result = mapOpt (\x->x+1) o1
```

Constructor Classes

- All map functions share the same structure

```
map     :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
mapOpt  :: (a -> b) -> Opt a -> Opt b
```

- They can all be written as:

```
fmap:: (a -> b) -> g a -> g b
```

  - where **g** is:

    [-] for lists, **Tree** for trees, and **Opt** for options

- Note that **g** is a function from types to types, i.e. a **type constructor**

Constructor Classes

- Capture this pattern in a constructor class,

```
class Functor g where
  fmap :: (a -> b) -> g a -> g b
```

A type class where the predicate is over type constructors

Constructor Classes

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)

instance Functor Opt where
  fmap f (Some s) = Some (f s)
  fmap f None = None
```

Constructor Classes

- Or by reusing the definitions map, mapTree, and mapOpt:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Opt where
  fmap = mapOpt
```

Constructor Classes

- We can then use the overloaded symbol **fmap** to map over all three kinds of data structures:

```
*Main> fmap (\x->x+1) [1,2,3]
[2,3,4]
it :: [Integer]

*Main> fmap (\x->x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
it :: Tree Integer

*Main> fmap (\x->x+1) (Some 1)
Some 2
it :: Opt Integer
```

- The `Functor` constructor class is part of the standard Prelude for Haskell