A Taste of Infinity
Infinite Lists

Lists in Haskell need not be finite. E.g.:

```haskell
list1 = [1..]       -- [1,2,3,4,5,6,...]
f x = x : (f (x+1))
list2 = f 1         -- [1,2,3,4,5,6,...]
list3 = 1:2:list3   -- [1,2,1,2,1,2,...]
```
Working with Infinite Lists

- Of course, if we try to perform an operation that requires consuming all of an infinite list (such as printing it or finding its length), our program will never yield a result.

- However, a program that only consumes a finite part of an infinite list will work just fine.

  \textit{take 5} [10..] \implies [10,11,12,13,14]
Lazy Evaluation

- The feature of Haskell that makes this possible is *lazy evaluation*.
- Only the portion of a list that is actually needed by other parts of the program will actually be constructed at run time.
- We will discuss the mechanics of lazy evaluation in much more detail later in the course.
More About Higher-Order Functions

(SOE Chapter 9)
Multi-Arg Functions in Haskell

What is the difference between

\[ f \ x \ y = x*y+5 \]

and

\[ f (x,y) = x*y+5 \]
Multi-Arg Functions in Haskell

\[ f :: \text{Integer} \to \text{Integer} \to \text{Integer} \]
\[ f \ x \ y = x \times y + 5 \]

\[ f :: (\text{Integer, Integer}) \to \text{Integer} \]
\[ f \ (x,y) = x \times y + 5 \]
Multi-Arg Functions in Haskell

When we write

\[ f :: \text{Integer} \to \text{Integer} \to \text{Integer} \]

what we really mean is:

\[ f :: \text{Integer} \to (\text{Integer} \to \text{Integer}) \]
Multi-Arg Functions in Haskell

The observation that an \textit{n-argument function} can equivalently be considered as a \textit{1-argument function} that returns an \textit{(n-1)-argument function} is called \textit{Currying} (after the great early-20th-century logician \textit{Haskell B. Curry}!)
Use of Currying

\[\text{listSum}, \text{listProd} :: [\text{Integer}] \rightarrow \text{Integer}\]
\[\text{listSum } xs = \text{foldr } (+) 0 xs\]
\[\text{listProd } xs = \text{foldr } (*) 1 xs\]

\[\text{and}, \text{or} :: [\text{Bool}] \rightarrow \text{Bool}\]
\[\text{and } xs = \text{foldr } (&&) \text{ True } xs\]
\[\text{or } xs = \text{foldr } (||) \text{ False } xs\]
Be Careful Though ...

Consider:
\[ f \ x = g \ (x+2) \ y \ x \]
This is not the same as:
\[ f = g \ (x+2) \ y \]
because the remaining occurrence of \( x \) becomes unbound. (Or, in fact, it might be bound by some outer definition!)

In general:
\[ f \ x = e \ x \]
is the same as
\[ f = e \]
only if \( x \) does not appear free in \( e \).
Simplifying Definitions

Recall:

```
reverse xs = foldl revOp [] xs
  where revOp acc x = x : acc
```

In the prelude we have: `flip f x y = f y x`. (what is its type?) Thus:
```
revOp acc x = flip (:) acc x
```
or even better:
```
revOp = flip (:)
```

And thus:
```
reverse xs = foldl (flip (:)) [] xs
```
or even better:
```
reverse = foldl (flip (:)) []
```
Anonymous Functions

- So far, all of our functions have been defined using an equation, such as the function `succ` defined by:
  \[ \text{succ } x = x+1 \]

- This raises the question: Is it possible to define a value that behaves just like `succ`, but has no name? Much in the same way that 3.14159 is a value that behaves like pi?

- The answer is yes, and it is written \( \backslash x \rightarrow x+1 \). Indeed, we could rewrite the previous definition of succ as:
  \[ \text{succ } = \backslash x \rightarrow x+1 \]
Sections

- Sections are like currying for infix operators. For example:
  
  \(+5\) = \x \to \ x + 5
  
  \(4-\) = \y \to \ 4 - y

  So in fact `succ` is just \(+1\)!

- Note the section notation is consistent with the fact that `(+)` is equivalent to \(\x \to \ \y \to \ x+y\).

- Although convenient in many situations, sections are less expressive than anonymous functions. For example, it’s hard to represent \(\x \to \ (x+1)/2\) as a section.

- You can also pattern match using an anonymous function, as in \(\(x:xs\) \to \ x\), which is the `head` function.
Function Composition

- Very often we would like to combine the effect of one function with that of another. *Function composition* accomplishes this for us, and is easily defined as the infix operator `(.)`:

  \[(f \ . \ g) \ x = f \ (g \ x)\]
  \[\text{-- i.e.:} \ (.) \ f \ g \ x = f \ (g \ x)\]

- So \(f.g\) means the same thing as \(\lambda x \rightarrow f \ (g \ x)\).

- Function composition can be used to simplify some of the previous definitions:

  \[
  \text{totalSquareArea sides} \\
  = \text{sumList} \ (\text{map squareArea sides}) \\
  = (\text{sumList} \ . \ \text{map squareArea}) \ \text{sides}
  \]

  Combining this with currying simplification yields:

  \[
  \text{totalSquareArea} = \text{sumList} \ . \ \text{map squareArea}
  \]
Qualified Types

(SOE Chapter 12)
Motivation

What should the principal type of (+) be?

- \textbf{Int -> Int -> Int} -- too specific
- \textbf{a -> a -> a} -- too general

It seems like we need something “in between”, that restricts “a” to be from the set of all number types, say \textbf{Num = \{Int, Integer, Float, Double, etc.\}}.

The type \textbf{a -> a -> a} is really shorthand for \((\forall a) a -> a -> a\)

\textit{Qualified types} generalize this by qualifying the type variable, as in \((\forall a \in \text{Num}) a -> a -> a\), which in Haskell we write as \textbf{Num a => a -> a -> a}
Type Classes

- “Num” in the previous example is called a *type class*, and should not be confused with a type constructor or a value constructor.
- “Num T” should be read “T is a member of (or an instance of) the type class Num”.
- Haskell’s type classes are one of its most innovative features.
- This capability is also called “overloading”, because one function name is used for potentially very different purposes.
- There are many pre-defined type classes, but you can also define your own.
Example: Equality

- Like addition, equality is not defined on all types (how would we test the equality of two functions, for example?).
- So the equality operator (==) in Haskell has type `Eq a => a -> a -> Bool`. For example:
  
  ```
  42 == 42 ➔ True
  'a' == 'a' ➔ True
  'a' == 42 ➔ << type error! >>
  (+1) == (\x->x+1) ➔ << type error! >>
  (types don’t match)
  ((->) is not an instance of Eq)
  ```

- Note: the type errors occur at compile time!
Equality, cont’d

- Eq is defined by this *type class declaration*:

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

- The last two lines are *default methods* for the operators defined to be in this class.

- A type is made an instance of a class by an *instance declaration*. For example:

  ```haskell
  instance Eq Int where
    x == y = intEq x y -- primitive equality for Ints
  instance Eq Float where
    x == y = floatEq x y -- primitive equality for Floats
  ```
Equality, cont’d

- **User-defined** data types can also be made instances of `Eq`. For example:

  ```hs
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  instance Eq (Tree a) where
    Leaf a1 == Leaf a2 = a1 == a2
    Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
    _ == _ = False
  ```

- But something is strange here: is “`a1 == a2`” on the right-hand side correct? How do we know that equality is defined on the type “`a`”???
Equality, cont’d

- **User-defined** data types can also be made instances of `Eq`. For example:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
  Leaf a1 == Leaf a2 = a1 == a2
  Branch l1 r1 == Branch l2 r2 = l1 == l2 && r1 == r2
  _         == _         = False
```

- But something is strange here: is “`a1 == a2`” on the right-hand side correct? How do we know that equality is defined on the type “`a`”???

- Answer: Add a constraint that requires `a` to be an equality type.
Consider this function:

\[
\begin{align*}
  x \text{ `elem` } [] &= \text{False} \\
  x \text{ `elem` } (y:ys) &= x == y \ || \ x \text{ `elem` } ys
\end{align*}
\]

Note the use of \((==)\) on the right-hand side of the second equation. So the principal type for `elem` is:

\[
\text{elem :: Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

This is inferred automatically by Haskell, but, as always, it is recommended that you provide your own type signature for all top-level functions.
Classes for Regions

- Useful slogan:
  
  “**polymorphism** captures similar structure over different values, while **type classes** capture similar operations over different structures.”

- For a simple example, recall from Chapter 8:

  ```
  containsS :: Shape -> Point -> Bool
  containsR :: Region -> Point -> Bool
  ```

- These are similar ops over different structures. So:

  ```
  class PC t where
      contains :: t -> Point -> Bool
  instance PC Shape where
      contains = containsS
  instance PC Region where
      contains = containsR
  ```
Haskell’s numeric types are embedded in a very rich, hierarchical set of type classes. For example, the **Num** class is defined by:

```haskell
class (Eq a, Show a) => Num a  where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

...where **Show** is a type class whose main operator is

```
show :: Show a => a -> String
```

See the Numeric Class Hierarchy in the Haskell Report on the next slide.
Haskell’s Standard Class Hierarchy
Coercions

- Note this method in the class **Num**:
  \[
  \text{fromInteger} :: \text{Num} \ a \Rightarrow \text{Integer} \rightarrow a
  \]

- Also, in the class **Integral**:
  \[
  \text{toIntInteger} :: \text{Integral} \ a \Rightarrow a \rightarrow \text{Integer}
  \]

- This explains the definition of **intToFloat**:
  \[
  \text{intToFloat} :: \text{Int} \rightarrow \text{Float}
  \]
  \[
  \text{intToFloat} \ n = \text{fromInteger} \ (\text{toIntInteger} \ n)
  \]

- These generic coercion functions avoid a quadratic blowup in the number of coercion functions.

- Also, every integer literal, say “42”, is really shorthand for “\text{fromInteger} 42”, thus allowing that number to be typed as *any* member of **Num**.
Derived Instances

- Instances of the following type classes may be automatically derived:
  - `Eq`, `Ord`, `Enum`, `Bounded`, `Ix`, `Read`, and `Show`
- This is done by adding a `deriving` clause to the `data` declaration. For example:

  ```ghc
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving (Show, Eq)
  ```

- This will automatically create an instance for `Show (Tree a)` as well as one for `Eq (Tree a)` that is precisely equivalent to the one we defined earlier.
Derived vs. User-Defined

Suppose we define an implementation of finite sets in terms of lists, like this:

```haskell
data Set a = Set [a]

insert (Set s) x = Set (x:s)

member (Set s) x = elem x s

union (Set s) (Set t) = Set (s++t)
```
Derived vs. User-Defined

We can automatically derive an equality function just by adding “deriving Eq” to the declaration.

```haskell
data Set a = Set [a]
    deriving Eq

insert (Set s) x = Set (x:s)

member (Set s) x = elem x s

union (Set s) (Set t) = Set (s++t)
```

But is this really what we want??
Derived vs. User-Defined

- No!
- E.g.,

\[(\text{Set } [1,2,3]) \equiv (\text{Set } [1,1,2,2,3,3]) \Rightarrow \text{False}\]
A Better Way

data Set a = Set [a]

instance Eq a => Eq (Set a) where
  s == t  =  subset s t && subset t s

subset (Set ss) t = all (member t) ss
Warning…

- The terminology used in Haskell (classes, instances, inheritance, etc.) is obviously intended to have something to do with Object-Oriented Programming.
- However, the exact correspondence is a bit tricky.
- I recommend *not* trying to think about this for the time being.
Reasoning About Type Classes

- Most type classes implicitly carry a set of laws.
- For example, the Eq class is expected to obey:
  \[(a /= b) = \text{ not } (a == b)\]
  \[(a == b) \&\& (b == c) \supseteq (a == c)\]
- Similarly, for the Ord class:
  \[a \leq a\]
  \[(a \leq b) \&\& (b \leq c) \supseteq (a \leq c)\]
  \[(a \leq b) \&\& (b \leq a) \supseteq (a == b)\]
  \[(a /= b) \supseteq (a < b) \lor (b < a)\]
- These laws capture the properties of an equivalence class and a total order, respectively.
- Unfortunately, there is nothing in Haskell that enforces the laws – it’s up to the programmer!