Advanced Programming
Handout 4

A Taste of Infinity

Infinite Lists

- Lists in Haskell need not be finite. E.g.:
  - 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.
  - take 5 [10..]  ➔ [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

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

and

\[ f (x,y) = x \cdot y + 5 \]

? 

Multi-Arg Functions in Haskell

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

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

\[ f :: (\text{Integer},\text{Integer}) \to \text{Integer} \]

\[ f (x,y) = x \cdot y + 5 \]

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

Use of Currying

\[ \text{listSum, listProd :: [Integer] \to Integer} \]

\[ \text{listSum \hspace{1em} xs = foldr (+) 0 \hspace{1em} xs} \]

\[ \text{listProd \hspace{1em} xs = foldr (*) 1 \hspace{1em} xs} \]

\[ \text{listSum} = \text{foldr} (+) 0 \]

\[ \text{listProd} = \text{foldr} (*) 1 \]

\[ \text{and, or :: [Bool] \to Bool} \]

\[ \text{and \hspace{1em} xs = foldr (\&\&) True \hspace{1em} xs} \]

\[ \text{or \hspace{1em} xs = foldr (||) False \hspace{1em} xs} \]

\[ \text{and} = \text{foldr} (\&\&) \text{ True} \]

\[ \text{or} = \text{foldr} (||) \text{ False} \]

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:

\[ \text{reverse } xs = \text{foldl } \text{revOp } [] \, xs \]
where \( \text{revOp } acc \, x = x : acc \)

In the prelude we have: \( \text{flip f x y = f y x} \). (what is its type?) Thus:

\[ \text{revOp } acc \, x = \text{flip } (:) \, acc \, x \]
or even better:

\[ \text{revOp} \]

And thus:

\[ \text{reverse } xs = \text{foldl} \, (\text{flip } (::)) \, [] \, xs \]
or even better:

\[ \text{reverse} \]

Anonymous Functions

- So far, all of our functions have been defined using an equation, such as the function \( \text{succ} \) defined by:

\[ \text{succ } x = x + 1 \]

- This raises the question: Is it possible to define a value that behaves just like \( \text{succ} \), 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 \( \lambda x \rightarrow x + 1 \). Indeed, we could rewrite the previous definition of \( \text{succ} \) as:

\[ \text{succ} \]

Sections

- Sections are like currying for infix operators. For example:

\[ (+5) = \lambda x \rightarrow x + 5 \]
\[ (4-) = \lambda y \rightarrow 4 - y \]

So in fact \( \text{succ} \) is just \((+1)\)!

- Note the section notation is consistent with the fact that \((+)\), for example, is equivalent to \( \lambda x \rightarrow \lambda y \rightarrow x + y \).

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

- You can also pattern match using an anonymous function, as in \( \lambda (x:xs) \rightarrow x \), 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 \((.)\):

\[ (f \cdot g) \, x = f \, (g \, x) \]

-- i.e.: \((.) \, f \, g \, x = f \, (g \, x)\)

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

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

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

Combining this with currying simplification yields:

\[ \text{totalSquareArea } = \text{sumList } \cdot \text{map } \text{squareArea} \]

Qualified Types

(SOE Chapter 12)

Motivation

- What should the principal type of \((+)\) be?
  - \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \) -- too specific
  - \( \text{a} \rightarrow \text{a} \rightarrow \text{a} \) -- too general

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

- The type \( \text{a} \rightarrow \text{a} \rightarrow \text{a} \) is really shorthand for \( \forall \text{a} \, a \rightarrow a \rightarrow a \)

- Qualified types generalize this by qualifying the type variable, as in \( \forall \text{a} \, (\text{a} \in \text{Num}) \, a \rightarrow a \rightarrow a \), which in Haskell we write as \( \text{Num} \, a \Rightarrow a \rightarrow a \rightarrow a \)
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:
  ```haskell```
  ```rml```
  
- Note: the type errors occur at compile time!

Equality, cont’d

- Eq is defined by this type class declaration:
  ```haskell```
  ```rml```

- 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```
  ```rml```

Equality, cont’d

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

Constraints / Contexts are Propagated

- Consider this function:
  ```haskell```
  ```rml```

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

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

Equality, cont’d

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

- 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.
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:
  ```haskell```
  ```haskell```
  ```haskell```
  ```haskell```

These are similar ops over different structures. So:
```haskell```

```
```

Coercions

- Note this method in the class `Num`:
  ```haskell```
- Also, in the class `Integral`:
  ```haskell```
- This explains the definition of `intToFloat`:
  ```haskell```
- These generic coercion functions avoid a quadratic blowup in the number of coercion functions.
- Also, every integer literal, say "42", is really shorthand for `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:
  ```haskell```
- This is done by adding a `deriving` clause to the `data` declaration. For example:
  ```haskell```
- 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```
- Insert:
  ```haskell```
- Member:
  ```haskell```
- Union:
  ```haskell```
**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??

**A Better Way**

```haskell
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
```

**Haskell Classes <> OO Classes**

- 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:
  
  \[
  (a /= b) = \text{not} (a == b) \\
  (a == b) \land (b == c) \Rightarrow (a == c)
  \]

- Similarly, for the Ord class:
  
  \[
  a <= a \\
  (a <= b) \land (b <= c) \Rightarrow (a <= c) \\
  (a <= b) \land (b <= a) \Rightarrow (a == b) \\
  (a /= b) \lor (a < b) \lor (b < a)
  \]

- 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!