Intro to Haskell

Lecture Outline

- Haskell
  - Covered syntax, algebraic data types and pattern matching

  - Lazy evaluation
  - Static typing and static type inference
  - Type classes
  - Monads … and more

Lazy Evaluation

- Unlike Scheme (and most programming languages) Haskell does lazy evaluation, i.e., normal order reduction
  - It won't evaluate an expression until it is needed

> **f x = []** --- **f** takes **x** and returns the empty list

> **f (repeat 1)** --- **repeat** produces infinite list **[1,1…**

> **[]**

> **head ([1..])** --- **[1..]** is the infinite list of integers

> **1**

- Lazy evaluation allows work with infinite structures

Lazy Evaluation

> f x = x*x

> f (5+1)

--- evaluates to **(5+1)** * **(5+1)**

--- evaluates argument only when needed

> fun n = n : fun(n+1)

> head (fun 5)

: denotes "cons" : constructs a list with head **n** and tail **fun(n+1)**

- Exercise: write a function that returns the (infinite) list of prime numbers

Lazy Evaluation

- Exercise: write a function that returns the (infinite) list of prime numbers

Static Typing and Type Inference

- Unlike Scheme, which is dynamically typed, Haskell is statically typed!
- Unlike Java/C++ we don't have to write type annotations. Haskell infers types!

> let f x = head x in f True

- Couldn't match expected type '[a]' with actual type 'Bool'
- In the first argument of 'f', namely 'True'

  In the expression: f True …

Static Typing and Type Inference

- Recall **apply_n f n x:**

> apply_n f n x = if n==0 then x else apply_n f (n-1) (f x)

> apply_n ((+) 1) True 0

<interactive>:32:1: error:

• Could not deduce (Num Bool) arising from a use of 'apply_n'

from the context: Num t2

bound by the inferred type of it :: Num t2 => t2

at <interactive>:32:1-22

• In the expression: apply_n ((+) 1) True 0

In an equation for 'it': it = apply_n ((+) 1) True 0

Lecture Outline

- Haskell
  - Covered syntax, algebraic data types and pattern matching

  - Lazy evaluation
  - Static typing and static type inference
  - Type classes
  - Monads … and more

Generic Functions in Haskell

- We can generalize a function when a function makes no assumptions about the type:

```
const :: a -> b -> a
const x y = x

apply :: (a->b)->a->b
apply g x = g x
```

Generic Functions

-- List datatype

data List a = Nil | Cons a (List a)

- Can we write function **sum** over a list of **a**'s?

sum :: a -> List a -> a

sum n Nil = n

sum n (Cons x xs) = sum (n+x) xs

- No. **a** no longer unconstraint. Type and function definition imply that we can apply **+** on **a** but
  - **+** is not defined on **all types**!
  - Type error: **No instance for (Num a) arising from a use of '+'**

Haskell Type Classes

- Not to be confused with Java classes/interfaces
- Define a type class containing the arithmetic operators

```
class Num a where
  (==)   :: a -> a -> Bool
  (+)    :: a -> a -> a
  …
instance Num Int where
  x == y = ...
  ...
instance Num Float where
  …
```

Read: A type **a** is an instance of the type class **Num** if it provides “overloaded” definitions of operations **==**, **+**, …

Read: **Int** and **Float** are instances of **Num**

Generic Functions with Type Class

```
sum :: (Num a) => a -> List a -> a
sum n Nil = n
sum n (Cons x xs) = sum (n+x) xs
```

- One view of type classes: predicates
  - **(Num a)** is a predicate in type definitions
  - Constrains the types we can instantiate a generic function to specific types
- A type class has associated laws

Type Class Hierarchy

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

class (Eq a) => Ord where
  (<), (<=), (>), (>=)  :: a -> a -> Bool
  min, max              :: a -> a -> a
```

- Each type class corresponds to one concept
- Class constraints give rise to a hierarchy
- **Eq** is a superclass of **Ord**
  - **Ord** inherits specification of **(==)** and **(/=)**
  - Notion of “true subtyping”

Lecture Outline

- Haskell
  - Covered syntax, algebraic data types and pattern matching

  - Lazy evaluation
  - Static typing and static type inference
  - Type classes
  - Monads … and more

Monads

- One source: All About Monads (haskell.org)
- Another source: Scott's book
- A way to cleanly compose computations
  - E.g., **f** may return a value of type **a** or **Nothing**

  Composing computations becomes tedious:

  case (f s) of

  Nothing → Nothing

  Just m → case (f m) …
- In Haskell, monads encapsulate IO and other **imperative** features

An Example: Cloned Sheep

```
type Sheep = …
father :: Sheep → Maybe Sheep
father = ...
mother :: Sheep → Maybe Sheep
mother = …
```

(Note: a cloned sheep may have both parents, or not...)

```
maternalGrandfather :: Sheep → Maybe Sheep
maternalGrandfather s = case (mother s) of
                            Nothing → Nothing
                            Just m → father m
```

An Example

```
mothersPaternalGrandfather :: Sheep → Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing → Nothing
                                 Just m → case (father m) of
                                            Nothing → Nothing
                                            Just gf → father gf
```

- Tedious, unreadable, difficult to maintain
- Monads help!

The Monad Type Class

- Haskell's Monad class requires 2 operations, >>= (bind) and return

class Monad **m** where

  // >>= (the bind operation) takes a monad
    // **m a**, and a function that takes **a** and turns

  // it into a monad **m b**

  (>>=) :: **m a** → (**a** → **m b**) → **m b**

  // return encapsulates a value into the monad

  return :: **a** → **m a**

The **Maybe** Monad

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

- Cloned Sheep example:

```
mothersPaternalGrandfather s =
     (return s) >>= mother >>= father >>= father
```

(Note: if at any point, some function returns Nothing, Nothing gets cleanly propagated.)

The **List** Monad

- The List type is a monad!

li **>>=** f = concat (map f li)

return x = [x]

Note: concat::[[**a**]] → [**a**]

e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use **any f** s.t. **f::a→[b]**. **f** may yield a list of 0,1,2,… elements of type **b**, e.g.,
  - > **f x = [x+1]**
  - > **[1,2,3] >>= f** --- yields ?

The **List** Monad

parents :: Sheep → [Sheep]

parents **s** = MaybeToList (mother **s**) ++
MaybeToList (father **s**)

grandParents :: Sheep → [Sheep]

grandParents **s** = (parents **s**) >>= parents

The **do** Notation

- **do** notation is syntactic sugar for monadic bind

```
> f x = x+1
> g x = x*5
> [1,2,3] >>= (return . f) >>= (return . g)
```

Or

```
> [1,2,3] >>= \x->[x+1] >>= \y->[y*5]
```

Or, make encapsulated element explicit with **do**

```
> do { x <- [1,2,3]; y <- (\x->[x+1]) x; (\y->[y*5]) y }
```

List Comprehensions

```
> [ x | x <- [1,2,3,4] ]
[1,2,3,4]
> [ x | x <- [1,2,3,4], x `mod` 2 == 0 ]
[2,4]
> [ [x,y] | x <- [1,2,3], y <- [6,5,4] ]
[[1,6],[1,5],[1,4],[2,6],[2,5],[2,4],[3,6],[3,5],[3,4]]
```

List Comprehensions

- List comprehensions are syntactic sugar on top of the **do** notation!

[x | x <- [1,2,3,4]] is syntactic sugar for

do { x <- [1,2,3,4]; return x }

[[x,y] | x <- [1,2,3], y <- [6,5,4]] is syntactic sugar for

do { x <- [1,2,3]; y<-[6,5,4]; return [x,y] }

- Which in turn, we can translate into monadic bind…

So What’s the Point of the Monad…

- Conveniently chains (builds) computation

- Encapsulates “mutable” state. E.g., **IO**:

openFile :: FilePath -> IOMode -> IO Handle

hClose :: Handle -> IO () -- void

hIsEOF :: Handle -> IO Bool

hGetChar :: Handle -> IO Char

These operations break “referentially transparency”. For example, **hGetChar** typically returns different value when called twice in a row!

The End