Contents

Lambda Abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch. There are times when it is handy to just write a function “inline” The notation is

```
\ x -> e
```

where

-`x` is a variable -`e` is an expressions that usually mentions `x`

The notation reads as “the function taking `x` as input and returning `e` as a result”. `\` refers to the symbol lambda. We can have nested abstractions.

```
\ x -> \ y -> e
```

Read as "the function taking `x` as input and returning a function that takes `y` as input and returns `e` as a result"

There is syntactic sugar for nested abstractions:

```
\ x y -> e
```

The following definitions pair are equivalent:

```
srq n = n * n
sqr = \ n -> n * n

add x y = x+y
add = \ x y -> x+y
```

This notation is based on "lambda calculus"

Lambda Application

In general, and application of a lambda abstraction to an argument looks like

```
(\ x -> x + x) a
       ^--e--^
-- Applied:
(a+a)
```

The result is a copy of `e` where any free occurrence of `x` has been replaced by `a`

Defining new types

- **Type synonyms**
  - `type Name = String`
  - Haskell considers both String and Name to be exactly the same type
- **"Wrapped" types**
  - `newtype Name = N String`
  - If `s` is a value of type `String`, then `N s` is a value of type `Name`. Haskell cosiders `String` and `Name` to be different

- **Algebraic Data Types**
  - `data Name = Official String String | NickName String`
  - If `f`, `s` and `n` are values of type `String`, the `Official f s` and `NickName n` are different values of type `Name`
  - Example
    - `Official "" "Arvind"`
    - `NickName "Arvind"`

Type Synonyms

```
type MyName = ExistingType
```

Haskell considers both `MyName` and `ExistingType` to be exactly the same

- Advantages
  - Clearer code documentation
  - Can use all existing functions defined for `ExistingType`
- Disadvantages
  - Typechecker does not distinguish `ExistingType` from any type like `MyName` defined like this

```
type Name = String; (name :: Name) = "Andrew"
type Addr = String; (addr :: Addr) = "TCD"
name ++ addr -- is well-typed
```

"Wrapping" Existing Types

```
newtype NewName = NewCons ExistingType
```

If `v` is a value of type `ExistingType`, and `NewCons v` is a value of type `NewName`

- Advantages
  - Typechecker treats `NewName` and `Existing Type` as different and incompatible
  - Can use type-class system to specify special handling of `NewName`
  - No runtime penalties of time and space
- Disadvtanges
  - Needs to have explicit `NewCons` on the front of values
  - Need to pattern match on `NewCons v` to define functions
  - None of the functions defined for `ExistingType` can be used directly

Algebraic Data Types

```
data ADTName
  = DCon1 Type11 Type12 ... Type1k1
  | DCon2 Type21 Type22 ... Type2k2
  ...
  | DConn Typen1 Typen2 ... Typenkn
```

If `vi1, ... viki` are values of types `Typei1 ... Typeiki`, then `Dconi vi1, ... viki` is a value of type `ADTName`, and values built with different `Dconi` are always different

- Advantages
  - The only way to add genuinely *new types* to your program
- Disadvantages
  - As per `newtype` - the need to use the `Dconi`, data-constructors, and to pattern match
  - Runtime and space ovorhead
    * Like union type in C

Type Parameters

- The types defined useing `type`, `newtype` and `data` can have type parameters themselves
  - `type TwoList t = ([t], [t])`
  - `newType BiList t = BiList ([t], [t])`
  - `data ListPair t = LPair [t] [t]`

User-defined Datatypes (data)

enums

With the `data` keyword, we can easily define new *enumerated* types

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

```
weekend :: Day -> Bool
weekend Saturday = True
weekend Sunday = True
weekend _ = False
```

We can define operations on values of this type by *pattern mathing*

Recursive structures

Haskell also allows data types to be define *recursively*.

We are familiar by now with lists in Haskell: writing the list `[1, 2, 3]` is just a shorthand for writing `1:2:3:[]`.

If lists were not built-in, we could define them with `data`

```
data List = Empty | Node Int List
```

compare

```
typedef struct {
    node* next;
    int value
} node;
```

Using this definition the list $< 1, 2, 3 >$ would be written

```
Node 1 (Node 2 (Node 3 Empty))
```

Recursive types usually mean recursive functions

```
length :: List -> Integer
length Empty = 0
length (Node _ rest) = 1 + (length rest)
```

Parameterised Data Types

Of course, those lists are not as flexible as the built-in lists, because they are not *polymorphic*. We can fix that by introducing a *type-variable*

```
data List t = Empty | Node t (List t)
```

compare:

`C++ class Node<T> { Node<T> *next; T valuel }`

No hange to the length function, but the type becomes

```
length :: (List a) -> Integer
```

What's in Name?

Consider the following `data` declaration

```
data MyType = AToken | ANum Int | AList [Int]
```

- The name `MyType` after the `data` keyword it the *type* name
- The names `AToken`, `ANum`, and `AList` on the rhs are *data-constructor* names
- Type names and data-constructor names are in different namespaces so they can overlap, e.g.:

```
data Thing = Thing String | Thang Int
```

- The same principle applies to newtypes:

```
newtype Nat = Nat Int
```

- We call the **Algebraic Datatypes** (ADTs)

Multiply-parameterised data Types

- Here is a useful data type

```
data Pair a b = Pair a b

divmod :: Integer -> Integer -> (Pair Integer Integer)
divmod x y = Pair (x/y) (x `mod` y)
```

Actually, list lists, "tuples" (of various sizes) are built in to Haskell and have a convenient syntax:

```
divmod :: Integer -> Integer -> (Integer, Integer)
divmod x y = (x / y, x `mod` y)
```

As you would expect, we can use pattern matching to open up the tuple:

```
f (x, y, z) = x + y + z
```

Data Types in Prelude

```
data () = () -- Not legal; for illustration
data Bool = False | True
data Char = ... 'a' | 'b' ... -- Unicode values
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Ordering = LT | EQ | GT
data [a] = [] | a : [a] -- Not legal; for illustration

data IO a = ... -- abstract; system/compiler dependant
data (a, b) = (a, b)
data (a, b, c) = (a, b, c) -- Not legal; for illustration
data IOError -- internet system dependent

data Int = minBound ... -1 | 0 | 1 ... maxBound
data Integer = ... -1 | 0 | 1 ...
data Float = ...
data Double = ...
```

Failure

A type that is often used in Haskell is one to model failure. While we can write functions such as `head` so that they fail outright

```
head (x:xs) = x
```

It is sometimes useful to model failure in a more management way

```
data Maybe a = Nothing
             | Just a
```

- Every Maybe value represents either a success or failure

```
mhead :: [a] -> Maybe a
mhead [] = Nothing
mhead (x:xs) = Just x
```

This technique is so common that `Maybe` and some useful functions are included in the standard Prelude