Declaring Types and Classes

COMP2209 - Programming III

Dr Julian Rathke

These slides use modified content from Graham Hutton’s slides for the module textbook Programming in Haskell
Type Synonyms

• What is the difference between the type String and [Char] in Haskell?
  • Not much at all, String is just another name, or a type synonym for [Char]
• So what is the point of String?
• It provides “better” documentation of the type of a function.
• For example, the function to tokenise a string would be
  
  \[
  \text{tokenise} :: \text{String} \to [\text{String}]
  \]
  which looks nicer than
  
  \[
  \text{tokenise} :: [\text{Char}] \to [ [\text{Char}]]
  \]
  and indicates intended usage.
Your own Type Synonyms

• Haskell allows you to write your own type synonyms.
• We simply use the keyword type along with the new name and the type it refers to;
• For example:

\[
\text{type Pos} = (\text{Int},\text{Int})
\]
• This declares that the name Pos now refers to this type (Int,Int) and functions may now use this synonym in their type declarations.

\[
\text{origin} : \text{Pos} \\
\text{origin} = (0,0)
\]

\[
\text{moveLeft} : \text{Pos} \rightarrow \text{Pos} \\
\text{moveLeft} (x,y) = (x-1,y)
\]
Parameterised Type Synonyms

• We typically will use type synonyms as a means of describing a particular assembly of types.

• For example, an *association* is a list of pairs of some key type and some value type.

• To allow this Haskell allows type synonyms to carry parameters:

  ```haskell
type Assoc k v = [(k,v)]
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k==k']
```

• These can **not** be partially applied:

  ```haskell
type IntAssoc = Assoc Int
```

Nested and Recursive Type Synonyms

- Nested type synonyms are allowed

```haskell
type Pos = (Int,Int)
```

```haskell
type Translation = Pos -> Pos
```

- Recursive type synonyms are **not** allowed

```haskell
type Tree = (Int, [Tree])
```

- It is important to notice that no new types are introduced using the type keyword. It really is just another way of writing down existing types.
Making new types

• Suppose I want to create a type Foo that behaves much like an Int yet I don’t want any functions that accept Foo to also accept an Int.

• e.g. Maybe Foo values have all been checked for some safety property such as being within a certain range.

• If I just use a type synonym and a function that uses a Foo there is no way to stop useFoo also accepting an Int

• What I will need is a way of distinguishing an Int value and a Foo value. We solve this problem by introducing **constructors** in to the types.

```haskell
type Foo = Int

useFoo :: Foo → Bool
useFoo n = ...
```
Simple Constructors

• A constructor is a name tag that identifies the values in a type.

• Constructors may be nullary (i.e. just names that exist in a type:
  • For example, Bool contains nullary constructors True and False
  • We can think of these as the basic building blocks of the type.

• Or constructors may take one (unary) or more arguments
  • We build larger values in the type using these.

• Constructor names are arbitrary but must begin with an upper case letter.
Keyword : data

- The keyword data allows us to define a new type.
- This is not a new name for an existing type:
  - It has different values to other types.
  - These values may only differ by a constructor name but they do differ.
- For example:

  ```haskell
  data Foo = F Int
  :type F 5
  F 5 :: Foo
  :type 5
  Num p => p
  ```

- Functions that accept a Foo type will not accept an Int.
- This helps enforce a key **Type Safety** property that functions do not accept values that they are not designed to accept.
Brand new types

• Of course we can use data to create entirely new finite types where the values are listed explicitly.

• For example:

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

• Notice the “or” separator | that allows us to write the possible different values that live in the type.

• We can define functions over this type by pattern matching the values.

```haskell
isWeekend :: Day → Bool
isWeekend Sat = True
isWeekend Sun = True
isWeekend _ = False

isWeekday :: Day → Bool
isWeekday = not.isWeekend
```
Constructors with Multiple parameters

• The constructors in a new type may take multiple parameters.
• These are just listed as per function application:

```
data Shape = Circle Float
            | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Interestingly, the constructors are actually functions here:

```
Circle :: Float -> Shape
Rect :: Float -> Float -> Shape
```
Data declaration parameters

• We’ve seen examples of constructors having (multiple) parameters.
• It is possible to parameterise the actual data type also.
• A heavily used type in Haskell is the Maybe type

\[
\text{data Maybe } a = \text{Nothing} \mid \text{Just } a
\]

• Values of this type are either just the nullary constructor Nothing or a value of type \( a \) tagged with Just
• This lets us take any datatype and add an extra value in the type to represent an error case, say.

\[
\begin{align*}
\text{safeHead} & : [a] \to \text{Maybe } a \\
\text{safeHead } [] & = \text{Nothing} \\
\text{safeHead } xs & = \text{Just } (\text{head } xs)
\end{align*}
\]

Any function that uses the output of safeHead must check whether a Nothing or Just value is returned - SAFETY!
Aside: newtype

- Declaring a brand new type for something like Foo above has computational cost:
  - Every function that uses a Foo must pattern match the F constructor, remove that to get at the Int it uses and then build any result back with F to get a Foo value again.
  - So it would be good to be able to be as efficient as just using an Int here but retain the type safety aspect of enforcing a particular version of Int to be used.
- The newtype declaration does just that.
- It can only be used for types of a single unary constructor
- The compiler will optimise away the constructor after the type safety is enforced

newtype Foo = F Int
F 5 :: Foo
Recursive Types

- Consider how one might define the type of Lists in Haskell:

  ```haskell
data MyList a = Empty | Cons a ( ??? )
```

- You would need to define a List type in terms of itself.
- That’s fine, Haskell allows recursively defined types:

  ```haskell
data MyList a = Empty | Cons a ( MyList a )
```

- Values in the MyList type are either a value called “Empty” or constructed from an `a` value and another List.
- We can define functions over MyList using pattern matching:

  ```haskell
dup Empty = Empty
dup (Cons m ms) = Cons m $ Cons m ms
```
Pattern Matching and Constructors

• What is clear from the above is that pattern matching in Haskell works on constructors - even user defined ones.

• This makes for a powerful mechanism for defining new types and functions over them.

• For example, we can easily build a datatype of Binary Trees

\[
\text{data Tree } a = \text{Leaf } a \mid \text{Node } (\text{Tree } a) \ a \ (\text{Tree } a)
\]

t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4) ) 5 (Node (Leaf 6) 7 (Leaf 9) )

Is an example value of type Tree Int

contains :: Eq a => a \rightarrow Tree a \rightarrow Bool
contains x (Leaf y) = x == y
contains x (Node l y r) =
\quad x==y \mid \text{contains } x \ l \mid \text{contains } x \ r

flatten :: Tree a \rightarrow [a]
flatten (Leaf y) = [y]
flatten (Node l y r) =
\quad \text{flatten } l ++[y]+\text{flatten } r
Variations on Tree

- It is not hard to see that the choice of data type for Tree is not at all fixed.
- We could reasonably use any of the following instead.

```haskell
data LTree a = Leaf a | Node (LTree a) (LTree a)
```

No values in interior nodes

```haskell
data ITree a = Leaf | Node (ITree a) a (ITree a)
```

No values in leaf nodes

```haskell
data DTrees a b = Leaf a | Node (DTrees a b) b (DTrees a b)
```

Different value types in leaf and interior nodes

```haskell
data MTrees a = Node a [MTrees a]
```

Arbitrarily many children (including 0)
Writing functions on all Tree types

• What if I want to write a function that could work with any of the above tree types?
• As part of this function maybe I need to flatten the tree to extract all values from it.
• What would the type of my function be?
• Obviously we would use a class constraint here. We would need a class Tree that guarantees at least a flatten function.
• How would I define such a thing.
• Easy:

        class Tree a where
        flatten :: a → [a]

• But how do I actually make sure that each Tree type is an instance of this class?
Class definitions

Generally speaking we can write class definitions as

class CONSTRAINTS => NAME a where
  function :: type
  function :: type
  ...
  function = default definition
  function = default definition
  ...

For example

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

  min x y | x <= y = x
           | otherwise = y

  max x y | x <= y = y
           | otherwise = x
Declaring instances

- To declare that a type is an instance of a class you use the `instance` keyword.
- You must also provide implementations for all of the undefined methods.

```haskell
instance Ord Bool where
    False < True = True
    _     < _     = False
    b <= c = (b<c) || (b==c)
    b > c = c < b
    b >= c = c <= b
```

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

    min x y | x <= y = x
             | otherwise = y
    max x y | x <= y = y
             | otherwise = x
```

Notice that we would need `Bool` to be an instance of `Eq` also in order to do this.

Won’t that be a lot of work for a new type?
Instances of Tree

• To return to our Tree example:

```haskell
instance Tree LTree where
  flatten (Leaf a) = [a]
  flatten (Node l r) = flatten l ++ flatten r

instance Tree ITree where
  flatten (Leaf) = []
  flatten (Node l y r) = flatten l ++ [y] ++ flatten r

instance Tree MTree where
  flatten (Node a cs) = a : map flatten cs
```

Etc
Deriving Instances

• For new types that are built in a structurally simple way Haskell can automatically **derive** implementations of the basic classes.

• We use the keyword deriving to do this

• This can save a lot of work when default implementations are fine.

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
          deriving (Eq, Ord, Show, Read)
```

• Tells Haskell to just work out the implementations of the functions in all of those classes by following the structure of the type Day.

• For example, show in the Show class will simply print out the string name of any given value and < in the Ord class is defined such that Mon < Tue ... < Sun
Deriving Instances

Of course for type declarations with parameters, if the type is declared to derive a class, then all of its parameters must derive that class also.

```haskell
data Shape = Circle Float | Rect Float Float
  deriving (Eq,Ord,Show,Read)
```

Is fine because Float derives all of these classes already.

```haskell
data DocumentedFunction a = Doc String (a -> a)
  deriving Eq
```

Is not okay because the function type (a -> a) is not an instance of the Eq class and hence no default == implementation can be found.
YOUR QUESTIONS

Next Lecture:
Functions on Trees and Graphs