Coursework Support Lecture 1: Haskell Facilities for Programming In the Large

Henrik Nilsson

University of Nottingham, UK
This Lecture

Some Haskell facilities that are particularly helpful for large-scale programming:

- The Haskell module system
- Haskell overloading
- Labelled fields (Haskell’s “record” system)
A Haskell program consists of a set of *modules*.

A module contains definitions:
- functions
- types
- type classes

The top module is called `Main`:

```haskell
module Main where

main = putStrLn "Hello World!"
```
By default, only entities defined within a module are in scope. But a module can *import* other modules, bringing their definitions into scope:

```haskell
module A where
f1 x = x + x
f2 x = x + 3
f3 x = 7

module B where
import A

g x = f1 x * f2 x + f3 x
```
The Prelude

There is one special module called the *Prelude*. It is *imported implicitly* into every module and contains standard definitions, e.g.:

- **Basic types** (*Int*, *Bool*, *tuples*, [], *Maybe*, ...)
- **Basic arithmetic operations** (+, *, ...)
- **Basic tuple and list operations** (*fst*, *snd*, *head*, *tail*, *take*, *map*, *filter*, *length*, *zip*, *unzip*, ...)

(It is possible to explicitly exclude (parts of) the Prelude if necessary.)
Qualified Names (1)

The **fully qualified name** of an entity $x$ defined in module $M$ is $M.x$.

$$g \ x = A.f_1 \ x \ * \ A.f_2 \ x + f_3 \ x$$

*Note! Different from function composition!!!*
Always write function composition with spaces:

$$f \ . \ g$$

The module **name space** is **hierarchical**, with names of the form $M_1.M_2......M_n$. This allows related modules to be grouped together.
Qualified Names (2)

Fully qualified names can be used to resolve name clashes. Consider:

```haskell
module A where
  f x = 2 * x

module B where
  f x = 3 * x

module C where
  import A
  import B

  g x = A.f x + B.f x
```

Two *different functions* with the same *unqualified name* $f$ in scope in $C$. Need to write $A.f$ or $B.f$ to disambiguate.
Import Variations

Another way to resolve name clashes is to be more precise about imports:

- `import A (f1,f2)`: Only `f1` and `f2`
- `import A hiding (f1,f2)`: Everything but `f1` and `f2`
- `import qualified A`: All names from `A` imported fully qualified only.

Can be combined in all possible ways; e.g.:

- `import qualified A hiding (f1, f2)`
Export Lists

It is also possible to be precise about what is exported:

module A (f1, f2) where

Various abbreviations possible; e.g.:

• A type constructor along with all its value constructors
• Everything imported from a specific module
Haskell Overloading (1)

What is the type of (==)?

E.g. the following both work:

\[
1 == 2 \\
'a' == 'b'
\]

I.e., (==) can be used to compare both numbers and characters.
Haskell Overloading (1)

What is the type of (==)?

E.g. the following both work:

```
1 == 2
'a' == 'b'
```

I.e., (==) can be used to compare both numbers and characters.

Maybe (==) :: a -> a -> Bool?
Haskell Overloading (1)

What is the type of (==)?

E.g. the following both work:

```
1 == 2
'a' == 'b'
```

I.e., (==) can be used to compare both numbers and characters.

Maybe (==) :: a -> a -> Bool?

No!!! Cannot work uniformly for arbitrary types!
A function like the identity function

\[
id :: \text{a} \rightarrow \text{a} \quad \text{id} \ x = x
\]

is **polymorphic** precisely because it works uniformly for all types: there is no need to “inspect” the argument.
A function like the identity function

\[ \text{id} :: \text{a} \to \text{a} \quad \text{id} \; x = x \]

is \textit{polymorphic} precisely because it works uniformly for all types: there is no need to “inspect” the argument.

In contrast, to compare two “things” for equality, they very much have to be inspected, and an \textit{appropriate method of comparison} needs to be used.
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
- But to add properly, we must understand what we are adding
Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when domain infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind
- But to add properly, we must understand what we are adding
- Not every type admits addition
Idea:
Haskell Overloading (4)

Idea:

- Introduce the notion of a type class: a set of types that support certain related operations.
Haskell Overloading (4)

Idea:

- Introduce the notion of a type class: a set of types that support certain related operations.
- Constrain those operations to only work for types belonging to the corresponding class.
Haskell Overloading (4)

Idea:

- Introduce the notion of a **type class**: a set of types that support certain related operations.
- **Constrain** those operations to **only** work for types belonging to the corresponding class.
- Allow a type to be **made an instance of** (added to) a type class by providing **type-specific implementations** of the operations of the class.
class Eq a where
    (==) :: a -> a -> Bool

(==) is not a function, but a method of the type class Eq. It’s type signature is:

(==) :: Eq a => a -> a -> Bool

Eq a is a class constraint. It says that that the equality method works for any type belonging to the type class Eq.
Various types can be made instances of a type class like `Eq` by providing implementations of the class methods for the type in question:

```haskell
instance Eq Int where
  x == y = primEqInt x y

instance Eq Char where
  x == y = primEqChar x y
```
Instances of Eq (2)

Suppose we have a data type:

```haskell
data Answer = Yes | No | Unknown
```

We can make `Answer` an instance of `Eq` as follows:

```haskell
instance Eq Answer where
  Yes    == Yes    = True
  No     == No     = True
  Unknown == Unknown = True
  _      == _      = False
```
Instances of `Eq (3)`

Consider:

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

Can `Tree` be made an instance of `Eq`?
Instances of \textbf{Eq} (4)

Yes, for any type \( a \) that is already an instance of \textbf{Eq}:

\begin{verbatim}
instance (Eq a) => Eq (Tree a) where
    Leaf a1 == Leaf a2 = a1 == a2
    Node t1l t1r == Node t2l t2r = t1l == t2l
        && t1r == t2r
    _ == _ = False
\end{verbatim}
Derived Instances

Instance declarations are often obvious and mechanical. Thus, for certain *built-in* classes (notably `Eq`, `Ord`, `Show`), Haskell provides a way to automatically derive instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

```haskell
data Tree a = Leaf a
           | Node (Tree a) (Tree a)
deriving Eq
```
Type classes form a hierarchy. E.g.:

class Eq a => Ord a where
  (<=) :: a -> a -> Bool
...

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.
Haskell vs. OO Overloading (1)

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types.

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider `read`:

```haskell
read :: (Read a) => String -> a
```

Note: overloaded on the result type! A method that converts from a string to any other type in class `Read`!
Haskell vs. OO Overloading (2)

> let xs = [1,2,3] :: [Int]
> let ys = [1,2,3] :: [Double]
> xs
[1,2,3]
> ys
[1.0,2.0,3.0]
> (read "42" : xs)
[42,1,2,3]
> (read "42" : ys)
[42.0,1.0,2.0,3.0]
> read "'a'" :: Char
'a'
Implementation (1)

The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.

Thus, internally \((==)\) is a higher order function with three arguments:

\[
(==) \ eqF \ x \ y = eqF \ x \ y
\]
Implementation (2)

An expression like

```
1 == 2
```

is essentially translated into

```
(==) primEqInt 1 2
```
Some Standard Haskell Classes (1)

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

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

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

Quiz: What is the type of a numeric literal like 42?
42 :: Int? Why?
Suppose we need to represent data about people:

- Name
- Age
- Phone number
- Post code

One possibility: use a tuple:

```haskell
type Person = (String, Int, String, String)
henrik = ("Henrik", 25, "8466506", "NG92YZ")
```
Problems? Well, the type does not say much about the purpose of the fields! Easy to make mistakes; e.g.:

```haskell
getPhoneNumber :: Person -> String
getPhoneNumber (_, _, _, pn) = pn
```

or

```haskell
henrik = ("Henrik", 25, "NG92YZ", "8466506")
```
Can we do better? Yes, we can introduce a new type with **named fields**:

```haskell
data Person = Person {
    name :: String,
    age :: Int,
    phone :: String,
    postcode :: String
}
```

`deriving (Eq, Show)`
Labelled Fields (4)

Labelled fields are just “syntactic sugar”: the defined type really is this:

```haskell
data Person = Person String Int String String
```

and can be used as normal.

However, additionally, the field names can be used to facilitate:

- Construction
- Update
- Selection
- Pattern matching
Construction

We can construct data without having to remember the field order:

```python
henrik = Person {
    age = 25,
    name = "Henrik",
    postcode = "NG92YZ",
    phone = "8466506"
}
```
Update (1)

Fields can be “updated”, creating new values from old:

\[
\text{> henrik } \{ \text{ phone } = "1234567" \} \\
\text{Person } \{ \text{ name } = "Henrik", \text{ age } = 25, \text{ phone } = "1234567", \text{ postcode } = "NG92YZ" \}
\]

Note: This is a \textit{functional} “update”! The old value is left intact.
Update (2)

How does “update” work?

henrik { phone = "1234567" }

gets translated to something like this:

f (Person a1 a2 _ a4) =
   Person a1 a2 "1234567" a4

f henrik
Selection

We automatically get a selector function for each field:

- `name :: Person -> String`
- `age :: Person -> Int`
- `phone :: Person -> String`
- `postcode :: Person -> String`

For example:

```plaintext
> name henrik
"Henrik"
> phone henrik
"8466506"
```
Field names can be used in pattern matching, allowing us to forget about the field order and pick only fields of interest.

\[
\text{phoneAge (Person \{phone = p, age = a\}) = p ++ ": " ++ show a}
\]

This facilitates adding new fields to a type as most of the pattern matching code usually can be left unchanged.
Multiple Value Constructors (1)

data Being = Person {
    name :: String,
    age :: Int,
    phone :: String,
    postcode :: String
}

| Alien |
|       |
|       |
|       |
|       |
|       |
|       |

deriving (Eq, Show)
Multiple Value Constructors (2)

It is OK to have the same field labels for different constructors as long as their types agree.
Distinct Field Labels for Distinct Types

It is **not** possible to have the same field names for **different** types! The following does not work:

```haskell
data X = MkX { field1 :: Int }

data Y = MkY { field1 :: Int, field2 :: Int }
```

One work-around: use a prefix convention:

```haskell
data X = MkX { xField1 :: Int }

data Y = MkY { yField1 :: Int, yField2 :: Int }
```
Advantages of Labelled Fields

- Makes intent clearer.
- Allows construction and pattern matching without having to remember the field order.
- Provides a convenient update notation.
- Allows to focus on specific fields of interest when pattern matching.
- Addition or removal of fields only affects function definitions where these fields really are used.