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.
What is a Type?

You can think of a type as a name for a collection of related values.

For example, in Haskell the basic type `Bool` contains the two logical values: `True` and `False`.

and the type `Int` represents all integer values in the range $-2^{63}$ to $2^{63}-1$. 
Why Types?

Types play a hugely important role in Haskell and other functional programming languages. They form part of the error checking in the language.

Haskell is *strongly statically typed* which means that types are checked at compile type and the compiler will throw an error if it finds problems with the way typed values are being used.

For example, the compiler will reject the expression

```
True + 17
```

because the operator (+) expects to be applied to two numeric values. This is an example of a type error.
Types in Haskell

If evaluating an expression \( e \) would produce a value of type \( t \), then \( e \) has type \( t \), written:

\[
e :: t
\]

Every well formed expression has a type, which can be automatically calculated at compile time using a process called **type inference**. That is, the compiler calculates the type of each expression for you.

In GHCi, the `:type` command calculates the type of an expression, without evaluating it:

\[
> \text{repeat True} \\
[\text{True}, \text{True}, \text{True}, \ldots] \\
> :\text{type repeat True} \\
\text{repeat True :: [Bool]}
\]
Basic Types and Compound Types

- There are two flavours of type in Haskell
- Basic Types: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`
- Compound Types built from the basic types by combining them using `type operators`
- There are various type operators but the most commonly used ones are List Types, Function Types and Pair (or Tuple) Types.
A brief tour of the basic types - **Bool**

We’ve seen this already, and there are two values in this type:

- **True**
- **False**

There are the usual logical connectives that operate on boolean values:

- **&&** :: **Bool** → **Bool** → **Bool**
- **||** :: **Bool** → **Bool** → **Bool**
- **not** :: **Bool** → **Bool**

And there is the usual conditional expression:

\[
\text{if } \text{bexp} \text{ then } \text{exp1} \text{ else } \text{exp2}
\]

but this must have both the ‘then’ and the ‘else’ branches.
A brief tour of the basic types - **Char**

This is the type of character data. The values here are single letters, digits, special characters etc.
cf. Unicode

To form characters you use single quotes and enter the unicode character or its hex code:

- `'a' :: Char`
- `'5' :: Char`
- `'\x2588' :: Char`

There are loads of library functions to manipulate chars. e.g. To convert between codes and chars use:

- `ord :: Char → Int`
- `chr :: Int → Char`
A brief tour of the basic types - **String**

Although treated as such strings are not really a basic type. They are effectively just lists of chars.

To form strings you use double quotes and enter the unicode characters or hex code:

```
“hello” :: [Char]
“This is\neasy” :: [Char]
```

Functions that work on lists can be applied to strings:

```
> length “abcde”
5
```
A brief tour of the basic types - **numbers**

There are a variety of numeric types: Int, Integer, Float, Double etc

Int is fixed precision integer values in range: $-2^{63}$ to $2^{63}-1$

Integer is arbitrary precision integer values.

Float is fixed single precision (IEEE) real values.

Double is double precision (IEEE) real values.

There are lots of library functions to convert between the types e.g.

- `floor`, `ceiling`, `round :: Float → Int`
- `fromInt :: Int → Float`
Something to ponder

Haskell uses type inference to determine the type of each expression. So what should the type of the expression comprising just the number 5 be?
Compound Types - Lists

For any valid Haskell type \( T \), we can form the Haskell type \([ T]\) of lists of values of type \( T \). So \([ \ ]\) is a type operator in Haskell.

\[
[ \text{True, False, False} ] :: [ \text{Bool} ]
\]

\[
[ 'a', 'b', 'c', 'd' ] :: [ \text{Char} ]
\]

\[
[ [3, 5], [2, 4, 6], [], [4] ] :: [ [\text{Int}] ]
\]

Lists must contain values of the same type.
Lists can be of arbitrary length - not represented in the type.
Compound Types - List Values

To form values of list type we use the list value constructor called “cons”

\[
(\_ : \_ ) :: a \to [a] \to [a]
\]

This says, to build a list (of type \(a\)) we need an element of that type and a list of that type! This is a bit circular.

Of course though we can always use the empty list to get started

\[
[] :: [a]
\]

This is why, in general, lists are built as sequences of applications of the cons operator

\[
1 : 2 : 3 : []
\]

with \([1,2,3]\) just being a syntactic sugar for the above.
Compound Types - **Tuples**

Unlike lists, a **tuple** is a sequence of values of possibly different types:

- `(True, False) :: (Bool, Bool)`
- `(4, True, ’c’) :: (Int, Bool, Char)`

In general,

- `(T_1, T_2, \ldots, T_n)` is the type of n-tuples where values of this type are of the form `(v_1, v_2, \ldots, v_n)` and each `v_i` has type `T_i`.

The length of the sequence of values is represented in and fixed by the type. Tuples can be formed from any type e.g. tuples of tuples and tuples of lists.
Compound Types - Functions

Given any Haskell types \( T \) and \( U \) we can form the function type

\[
T \rightarrow U
\]

Values of this type are functions that take a value of type \( T \) and return a value of type \( U \).

Again, \( T \) and \( U \) can be any type, so we can use tuples to make functions of “multiple” arguments and results.

```
add :: (Int,Int) \rightarrow Int
add (x,y) = x+y

zeroto :: Int \rightarrow [Int]
zeroto n = [0..n]
```

Technically these are functions of a single argument / return type.
Curried Functions

We use the fact that functions can be built using any type to have functions that return functions. We call these Curried Functions.

\[
\text{add'} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

\[
\text{add'} x \ y = x + y
\]

Here \(\text{add'}\) takes an integer, \(n\) say, and returns a function. This function has type \(\text{Int} \rightarrow \text{Int}\) and it accepts a single integer, \(m\) say, and returns the value \((\text{add'} n \ m)\).

To compare: both \(\text{add}\) and \(\text{add'}\) evaluate to the same value when applied to \(n\) and \(m\) but \(\text{add}\) accepts both \(n\) and \(m\) together in a tuple whereas \(\text{add'}\) accepts them one-by-one.

\[
\text{add} :: \text{(Int,Int)} \rightarrow \text{Int}
\]

\[
\text{add'} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

Named after logician Haskell Curry
Curried Functions of more arguments

We can nest the currying of functions to obtain functions of multiple arguments.

\[
\text{mult} :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))
\]
\[
\text{mult} \ x \ y \ z = x*y*z
\]

We can apply mult to a single argument to obtain a curried function of type \(\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})\). This can subsequently be applied to another argument to return a function of type \(\text{Int} \rightarrow \text{Int}\).

\[
\text{let } f = \text{mult} \ 3 \ \text{in}
\]
\[
\text{let } g = f \ 4 \ \text{in}
\]
\[
g \ 5
\]

\(f\) takes two ints \(m, n\) and returns \(3 \cdot m \cdot n\)

\(g\) takes one int \(n\) and returns \(3 \cdot 4 \cdot n\)
Polymorphic Functions

Expressions in Haskell can often be potentially given many different types.

For example, the function `length` could have types

```
[Bool] → Int
[Int] → Int
[[Char]] → Int
```

To represent this situation we use type variables to stand for “any type” when writing a compound type:

```
length :: [a] → Int
```

We say that a function is a **polymorphic** if its type contains one or more type variables.
Many of the functions defined in the standard prelude are polymorphic.

For example

\[
\begin{align*}
\text{fst} & \:: \ (a,b) \rightarrow a \\
\text{head} & \:: \ [a] \rightarrow a \\
\text{take} & \:: \ \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{zip} & \:: \ [a] \rightarrow [b] \rightarrow [(a,b)] \\
\text{id} & \:: \ a \rightarrow a
\end{align*}
\]

Type variable names must begin with a lower-case letter.
Not quite polymorphic

• Consider the function `add x y = x + y`
• What type does `add` have?
• It can’t accept *any* type as it needs to use `+` on its arguments.
• However it could accept `Int`, `Integer`, `Float`, `Double` values.
• In general, it would be reasonable to allow `add` to take arguments of any type for which the `+` operator makes sense.
• We need a way to specify this somehow because type variables only allow for *any* type to be used.
• This is where class constraints enter the picture.

```
add :: Num a => a -> a -> a
```

Num is a type class, `add` accepts arguments of any type `a` that is an instance of the `Num` class.
Basic Type Classes

• Type classes are Haskell’s way of providing overloaded operations.

• Think of a Haskell class a bit like an interface in Java - it defines a named type and certain operations that anything implementing that type must support.

• Haskell types may be instances of more than one class.

• There are a number of basic built in type classes - let’s look at a few of these now.
The Eq and Ord classes

Class Eq represents types that support the \(==\) comparison operation. The operators listed in this class are

\[
(==) :: a \to a \to \text{Bool}
\]

\[
(/=) :: a \to a \to \text{Bool}
\]

Class Ord represents types that support order comparisons. The operators listed in this class are

\[
(<) :: a \to a \to \text{Bool}
\]

\[
(>) :: a \to a \to \text{Bool}
\]

\[
(\leq) :: a \to a \to \text{Bool}
\]

\[
(\geq) :: a \to a \to \text{Bool}
\]

\[
\text{min} :: a \to a \to a
\]

\[
\text{max} :: a \to a \to a
\]

All of the basic types (and lists and tuples built from these basic types) are instances of the Eq and Ord classes.
The Show and Read classes

This pair of complementary classes deal with converting types to and from the String type.

Class Show represents types that support the function

\[
\text{show} :: a \rightarrow \text{String}
\]

The idea is that any type that is an instance of Show has its own way of pretty printing values of that type (cf. toString in Java).

Class Read represents types that support the function

\[
\text{read} :: \text{String} \rightarrow a
\]

The idea is that any type that is an instance of Read has its own way of parsing values of that type from a string.

All of the basic types (and lists and tuples built from these basic types) are instances of the Show and Read classes.
The Number Classes

These classes are fairly self-explanatory. They are used to support the variety of number types that occur in Haskell.

Class Num represents types that represent numbers and common operations on them

\[
\begin{align*}
(+) & : \text{a} \to \text{a} \to \text{a} \\
(\ast) & : \text{a} \to \text{a} \to \text{a} \\
\text{abs} & : \text{a} \to \text{a}
\end{align*}
\]

Class Integral

\[
\begin{align*}
\text{div} & : \text{a} \to \text{a} \to \text{a} \\
\text{Mod} & : \text{a} \to \text{a} \to \text{a}
\end{align*}
\]

Class Fractional

\[
\begin{align*}
\text{negate} & : \text{a} \to \text{a} \\
\text{signum} & : \text{a} \to \text{a} \\
\text{recip} & : \text{a} \to \text{a}
\end{align*}
\]
Done pondering?

So what should the type of the expression comprising just the number 5 be?

5 :: Num a => a
YOUR QUESTIONS

Next Lecture:
Defining Functions