Chapter 5 – Introduction, The Hugs System, Types and Classes
The Software Crisis

• How can we cope with the size and complexity of modern computer programs?

• How can we reduce the time and cost of program development?

• How can we increase our confidence that the finished programs work correctly?
Programming Languages

• One approach to the software crisis is to design new programming languages that
  – Allow programs to be written clearly, concisely, and at a high-level of abstraction;
  – Support reusable software components;
  – Encourage the use of formal verification;
  – Permit rapid prototyping;
  – Provide powerful problem-solving tools.
What is a Functional Language?

• Functional languages provide a particularly elegant framework in which to address these goals.

• Opinions differ, and it is difficult to give a precise definition, but generally speaking:
  – Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
  – A functional language is one that supports and encourages the functional style.
Example

Summing the integers 1 to 10 in C or Java:

```java
total = 0;
for (i = 1; i ≤ 10; ++i)
    total = total+i;
```

The computation method is variable assignment.
Example

Summing the integers 1 to 10 in Haskell:

\[
\text{sum \ [1..10]}
\]

The computation method is function application.
Why Haskell?

• Very high level above machine architecture – powerful
• Functional language – everything returns a result
• Interactive – code and test immediately
• No side effects
• Native list operations
• Pattern matching programming style
Why Haskell?

- Useful for studying fundamentals of languages
- Used to implement language interpreter
- Good at handling complex data and combining components
- Not a high-performance language (prioritise programmer-time over computer-time).
A Taste of Haskell

\[
f [\ ] = []
\]
\[
f (x:xs) = f \ ys \ ++ \ [x] \ ++ \ f \ zs
\]

where
\[
y = [a \mid a \leftarrow xs, a \leq x]
\]
\[
z = [b \mid b \leftarrow xs, b > x]
\]
The Hugs System

• Hugs is an implementation of Haskell 2006, and is the most widely used Haskell system;

• The interactive nature of Hugs makes it well suited for teaching and prototyping purposes;

• Hugs is available on the web from:
  www.haskell.org/hugs
Starting Hugs

On a Unix system, Hugs can be started from the % prompt by simply typing `hugs`:
The Hugs > prompt means that the Hugs system is ready to evaluate an expression.

For example:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```
The Standard Prelude

The library file Prelude.hs provides a large number of standard functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

- Select the first element of a list:

```
> head [1,2,3,4,5]
1
```
- Remove the first element from a list:

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Select the last element of a list:

```
> last [1,2,3,4,5]
5
```

- Remove the last element of a list:

```
> init [1,2,3,4,5]
[1,2,3,4]
```
Check if a list is empty:

```haskell
> null [1,2,3,4,5]
False
```

Select the nth element of a list:

```haskell
> [1,2,3,4,5] !! 2
3
```

Select the first n elements of a list:

```haskell
> take 3 [1,2,3,4,5]
[1,2,3]
```
- Remove the first n elements from a list:

  ```
  > drop 3 [1,2,3,4,5]
  [4,5]
  ```

- Calculate the length of a list:

  ```
  > length [1,2,3,4,5]
  5
  ```

- Calculate the sum of a list of numbers:

  ```
  > sum [1,2,3,4,5]
  15
  ```
Calculate the product of a list of numbers:

```> product [1,2,3,4,5]
120```

Append two lists:

```> [1,2,3] ++ [4,5]
[1,2,3,4,5]```

Reverse a list:

```> reverse [1,2,3,4,5]
[5,4,3,2,1]```
Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

\[ f(a, b) + c \cdot d \]

Apply the function \( f \) to \( a \) and \( b \), and add the result to the product of \( c \) and \( d \).
In Haskell, function application is denoted using space, and multiplication is denoted using *.

\[ f \ a \ b \ + \ c*d \]

As previously, but in Haskell syntax.
Moreover, function application is assumed to have higher priority than all other operators.

\[ f \ a + b \]

Means \((f \ a) + b\), rather than \(f \ (a + b)\).
Examples

Mathematics

\( f(x) \)  \hspace{1cm}  \text{Haskell}  \hspace{1cm}  f \ x \\
\( f(x,y) \)  \hspace{1cm}  \hspace{1cm}  \hspace{1cm}  f \ x \ y \\
\( f(g(x)) \)  \hspace{1cm}  \hspace{1cm}  \hspace{1cm}  f \ (g \ x) \\
\( f(x,g(y)) \)  \hspace{1cm}  \hspace{1cm}  \hspace{1cm}  f \ x \ (g \ y) \\
\( f(x)g(y) \)  \hspace{1cm}  \hspace{1cm}  \hspace{1cm}  f \ x \ * \ g \ y \)
Haskell Scripts

• As well as the functions in the standard prelude, you can also define your own functions;

• New functions are defined within a **script**, a text file comprising a sequence of definitions;

• By convention, Haskell scripts usually have a `.hs` suffix on their filename. This is not mandatory, but is useful for identification purposes.
My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.

Start an editor, type in the following two function definitions, and save the script as test.hs:

double x = x + x
quadruple x = double (double x)
Leaving the editor open, in another window start up Hugs with the new script:

```% hugs test.hs```

Now both Prelude.hs and test.hs are loaded, and functions from both scripts can be used:

```> quadruple 10
40
> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]```
Leaving Hugs open, return to the editor, add the following two definitions, and resave:

\[
\text{factorial } n = \text{product } [1..n]
\]

\[
\text{average } ns = \text{sum } ns \ `\text{div}` \text{ length } ns
\]

Note:

- `div` is enclosed in back quotes, not forward;
- `x `f` y` is just syntactic sugar for `f x y`. 
Hugs does not automatically detect that the script has been changed, so a `reload` command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```
Naming Requirements

• Function and argument names must begin with a lower-case letter. For example:

  myFun  fun1  arg_2  x’

By convention, list arguments usually have an s suffix on their name. For example:

  xs  ns  nss
The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$

- $a = 10$
- $b = 20$
- $c = 30$
The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

\[
a = b + c
\text{ where }
\begin{align*}
b &= 1 \\
c &= 2
\end{align*}
\]
\[
d = a \times 2
\]

\[
a = b + c
\text{ where }
\begin{align*}
\{b &= 1; \\
c &= 2\}
\end{align*}
\]
\[
d = a \times 2
\]

implicit grouping

explicit grouping
# Useful Hugs Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:load  <em>name</em></td>
<td>load script <em>name</em></td>
</tr>
<tr>
<td>:reload</td>
<td>reload current script</td>
</tr>
<tr>
<td>:edit  <em>name</em></td>
<td>edit script <em>name</em></td>
</tr>
<tr>
<td>:edit</td>
<td>edit current script</td>
</tr>
<tr>
<td>:type  <em>expr</em></td>
<td>show type of <em>expr</em></td>
</tr>
<tr>
<td>:?</td>
<td>show all commands</td>
</tr>
<tr>
<td>:quit</td>
<td>quit Hugs</td>
</tr>
</tbody>
</table>
Exercises

(1) Try out the above slides using Hugs.

(2) Fix the syntax errors in the program below, and test your solution using Hugs.

\[
N = a \text{'div'} \text{length} \; xs \\
\text{where} \\
a = 10 \\
xs = [1,2,3,4,5]
\]
(3) Show how the library function `last` that selects the last element of a list can be defined using the functions introduced in this lecture.

(4) Can you think of another possible definition?

(5) Similarly, show how the library function `init` that removes the last element from a list can be defined in two different ways.
What is a Type?

A **type** is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False  True
```
Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.

> 1 + False
Error

1 is a number and False is a logical value, but + requires two numbers.
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.
All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

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

```hugs
> not False
True

> :type not False
not False :: Bool```
Basic Types

Haskell has a number of basic types, including:

- **Bool** - logical values
- **Char** - single characters
- **String** - strings of characters
- **Int** - fixed-precision integers
- **Integer** - arbitrary-precision integers
- **Float** - floating-point numbers
List Types

A list is sequence of values of the same type:

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

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

In general:

\[\text{[t]} \text{ is the type of lists with elements of type t.}\]
The type of a list says nothing about its length:

- \([\text{False, True}] :: [\text{Bool}]\)
- \([\text{False, True, False}] :: [\text{Bool}]\)

The type of the elements is unrestricted. For example, we can have lists of lists:

- \([[[\text{a'}],[\text{b'}, \text{c'}]]] :: [[[\text{Char}]]]\)
Tuple Types

A tuple is a sequence of values of different types:

(False,True)  ::  (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.
The type of a tuple encodes its size:

\[(\text{False}, \text{True}) \quad :: \quad (\text{Bool}, \text{Bool})\]

\[(\text{False}, \text{True}, \text{False}) \quad :: \quad (\text{Bool}, \text{Bool}, \text{Bool})\]

The type of the components is unrestricted:

\[('a', (\text{False}, 'b')) \quad :: \quad (\text{Char}, (\text{Bool}, \text{Char}))\]

\[(\text{True}, ['a', 'b']) \quad :: \quad (\text{Bool}, [\text{Char}])\]
Function Types

A function is a mapping from values of one type to values of another type:

\[
\text{not} :: \text{Bool} \to \text{Bool} \\
\text{isDigit} :: \text{Char} \to \text{Bool}
\]

In general:

\[ t_1 \to t_2 \] is the type of functions that map values of type \( t_1 \) to values to type \( t_2 \).
Note:

- The arrow $\rightarrow$ is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

  ```haskell
  add       :: (Int,Int) → Int
  add (x,y)  = x+y
  
  zeroto    :: Int → [Int]
  zeroto n  = [0..n]
  ```
Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

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

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

add’ takes an integer x and returns a function add’ x. In turn, this function takes an integer y and returns the result x+y.
Note:

- add and add’ produce the same final result, but add takes its two arguments at the same time, whereas add’ takes them one at a time:

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

- Functions that take their arguments one at a time are called **curried** functions, celebrating the work of Haskell Curry on such functions.
Functions with more than two arguments can be curried by returning nested functions:

\[
\text{mult} \quad :: \quad \text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})) \\
\text{mult} \ x \ y \ z = x \times y \times z
\]

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x \times y \times z.
Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:

\[
\begin{align*}
\text{add’ } 1 &:: \text{Int} \rightarrow \text{Int} \\
\text{take } 5 &:: \text{[Int]} \rightarrow \text{[Int]} \\
\text{drop } 5 &:: \text{[Int]} \rightarrow \text{[Int]}
\end{align*}
\]
Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

• The arrow \( \to \) associates to the right.

\[
\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
\]

Means \( \text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})) \).
As a consequence, it is then natural for function application to associate to the left.

\[ \text{mult } x \ y \ z \]

Means \(((\text{mult} \ x) \ y) \ z\).

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.
Polymorphic Functions

A function is called polymorphic (“of many forms”) if its type contains one or more type variables.

\[
\text{length} :: [a] \rightarrow \text{Int}
\]

for any type \( a \), length takes a list of values of type \( a \) and returns an integer.
Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2
```

```
> length [1,2,3,4]
4
```

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.
Many of the functions defined in the standard prelude are polymorphic. For example:

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

A polymorphic function is called overloaded if its type contains one or more class constraints.

```
sum :: Num a ⇒ [a] → a
```

for any numeric type `a`, `sum` takes a list of values of type `a` and returns a value of type `a`. 
Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> sum [1,2,3]
6
> sum [1.1,2.2,3.3]
6.6
> sum ['a','b','c']
ERROR
```

- `a = Int`
- `a = Float`
- Char is not a numeric type
Haskell has a number of type classes, including:

- **Num** - Numeric types
- **Eq** - Equality types
- **Ord** - Ordered types

For example:

\[ (+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a \]
\[ (==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]
\[ (<) :: \text{Ord } a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]
Hints and Tips

• When defining a new function in Haskell, it is useful to begin by writing down its type;

• Within a script, it is good practice to state the type of every new function defined;

• When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.
Exercises

(1) What are the types of the following values?

- `['a', 'b', 'c']`
- `('a', 'b', 'c')`
- `[(False, '0'), (True, '1')]`
- `([False, True], ['0', '1'])`
- `[tail, init, reverse]`
(2) What are the types of the following functions?

- second \( xs \) = head (tail \( xs \))
- swap (\( x,y \)) = (\( y,x \))
- pair \( x, y \) = (\( x,y \))
- double \( x \) = \( x \times 2 \)
- palindrome \( xs \) = reverse \( xs \) == \( xs \)
- twice \( f, x \) = \( f \) (\( f \) \( x \))

(3) Check your answers using Hugs.