COMP30026 Models of Computation

Introduction to Haskell

Harald Søndergaard

Lecture 2

Semester 2, 2017

Haskell

We will use Haskell to illustrate various concepts in this subject; Haskell will be delivered in the Grok environment that most of you are familiar with.

The Grok environment will allow you to go through introductions to different language features and to test your knowledge interactively.

This will mostly happen in tutes and in your own time.

I am making this slide set available as another resource, but I don't intend to go through it in detail in lectures.

Instead we will use a lecture to develop a Haskell program.

Functional Programming

Haskell is a functional programming language.

Functions are the basic building blocks from which programs are constructed.

The basic operation in a functional languages is function application.

ghc is a Haskell compiler (the Glasgow Haskell Compiler).

ghci is an interactive programming environment for Haskell. Within it you can create, modify and run programs.

Learning Haskell

Beyond Grok, there are lots of learning tools. For example, check out `tryhaskell.org/`.

Under Reading Resources, find (and download) the book by O'Donnell, Hall and Page. Chapter 1 is an introduction to Haskell.

More detailed information can be found at `www.haskell.org/`.

The Haskell 2010 Report has definitions of the language and libraries: `www.haskell.org/onlinereport/haskell2010/`.

Haskell Outside the Grok Environment

You will find Haskell on the Engineering student machines. It is also easy enough to download and install the Haskell Platform from `www.haskell.org/`.

To use ghci on Windows machines, find '`winghci`'.

On a Unix system, simply type '`ghci`'.

In the simplest mode of use, ghci is an interactive calculator:
You can type an expression and ghci responds with its value.

You can also edit and load programs (scripts) from within ghci.

Pure Functional Programming

Haskell is pure in that it insists on functions with no side-effects and no state.

This way functions form self-contained units with all connections to the outside world explicit—they are transparent building blocks.

Small-scale functions are combined to form larger-scale functions.

Larger-scale functions are combined in the same way to form even larger scale functions, until eventually we have a complete program.

Compared to most programming languages, Haskell has a small set of "rules" and an extremely small set of "exceptions" to its rules.

Manipulating Functions

Consider the simplest way to combine functions: take the result of one function application and make it the argument of another.

Suppose we have a function `square` to square numbers. We may then take a number n to the fourth power n^4 by squaring it and then squaring the result:

```
> let fourth_power n = square (square n)
>     in fourth_power 3
```

Or simpler, using the composition operation `.`:

```
> let fourth_power = square . square
>     in fourth_power 3
```

Higher Order Functions

Another way that functions can be combined is by passing functions as arguments to other functions.

If we have a function to square numbers, we can create a function to square each number in a list:

```
> square_all_nums xs = map square xs
```

The function `map`, which is easily defined, takes a function and a list and applies the function to each element in the list.

Functions which have other functions as arguments are called higher order functions. Such functions are very powerful programming tools.

Haskell Uses Clean Mathematical Notation

Haskell's syntax is derived from conventional mathematical notation.

However, in mathematics we write $f(1)$ and $g(3, 5)$.

Haskell programmers would usually write that `f 1` and `g 3 5`.

(The juxtaposition of expressions denotes function application.)

This notation was chosen to make definitions less cluttered and expressions easier to type.

Some Haskell functions are written in infix notation to match their mathematical counterparts:

```
2 + 3      6 * 5      (6 - 2) / 4
```

Haskell Features

- lexical scoping
- higher-order functions
- pattern matching
- lazy functions and data constructors
- polymorphism
- strong static type inference
- user-defined data types
- principled overloading
- monadic I/O
- type-safe modules

No loop constructs—iteration is expressed as recursion or via higher-order functions.

The Type System

Haskell types are polymorphic: types may contain type variables which are universally quantified over all types.

Haskell is strongly typed: every expression has exactly one "most general" (or principal) type. It is always possible to statically infer this type – user-supplied type signatures are optional (but recommended).

- Strong typing aids reasoning about programs.
- Polymorphism means very expressive types.
- All type errors are detected at "compile time."
- No run-time type tests are required.

Example Session

```
? 3^4
 81
? 15 `div` 7
 2
? 15 `mod` 7
 1
? 2 < 3
 True
? sqrt 4.0
 2.0
? sum [3,4,5]
 12
? length [3,4,5]
 3
```

A standard library of pre-defined functions, called the prelude, is always loaded.

Lexical Matters

Case matters – `foo`, `Foo` and `FOO` are different identifiers.

Identifiers used for bound variables and type variables start with lower case. Those used for types and constructors start with upper case.

Infix operators may be coerced into prefix functions and vice versa.

For example, `x + y` may be written `(+) x y`, and `f x y` may be written `x` ‘`f`‘ `y`.

Layout Sensitivity

Layout is significant in Haskell. Line-breaks matter!

Although they contain the same sequence of tokens,

```
> f = let a = 7 in (+) 2 a
> b = 42
```

and

```
> f = let a = 7 in (+) 2
> a b = 42
```

define different entities.

Indentation in Programs

All components of a definition have to be right of the = sign, or, with where clauses, must not be to the left of the first component. This way Haskell has no need for C-style separators. Instead of

```
> f a + f 1 where { a = 5;
>                   f x = x + 1 }
```

(which is legal Haskell) one may write

```
> f a + f 1 where a = 5
>                 f x = x + 1
```

Using indentation, you must be careful when writing definitions. Find a useful layout convention and stick to it.

Lists Are Pervasive in Haskell

Haskell supports list comprehensions:

```
> qs []     = []
> qs (x:xs) = qs [y | y <- xs, y<x]
>             ++ [x]
>             ++ [y | y <- xs, y==x]
>             ++ qs [y | y <- xs, y>x]
```

and arithmetic sequences:

```
[1..9]      -- [1,2,3,4,5,6,7,8,9]
[1,3..10]   -- [1,3,5,7,9]
[1..]       -- infinite list of positive integers
[1,3..]     -- infinite list of positive odds
```

List Operations

Some pre-defined operations on lists are `(:)`, `head`, `tail`, and `(++)`.

`(:)`, or "cons", is the list constructor; it takes an element `x` and a list `xs` and returns a list whose head (first element) is `x` and whose tail is `xs`.

The Prelude also defines `last`, `length`, `null`, `reverse`, `elem`, `notElem`, `zip`, `take`, and `(!!)`.

The last is a selector: `xs!!n` returns the $n+1$'th element of the list *xs*. (Numbering starts from 0.)

Other useful list functions are found in module `Data.List`, for example, `(\\)` (list difference), `nub`, `group`, `transpose`,...

Infinite Lists

Infinite data structures mix well with lazy evaluation.

```
> ones = 1 : ones
>
> nums_from n = n : nums_from (n+1)
>
> squares = [x^2 | x <- nums_from 0]
>
> fibs = 1:1:[a+b|(a,b) <- zip fibs (tail fibs)]
```

Elements are computed on demand, so "consumer-producer" control flow is automatic.

Using Infinite Lists

We can define the function `take` inductively:

```
> take _ []       = []
> take 0 _        = []
> take n (x:xs) = x : take (n-1) xs
```

Now `take 5 fibs` yields `[1,1,2,3,5]`.

This leads to a programming style void of arbitrary upper limits on numbers and sizes (of structures).

List Comprehensions

What does a list comprehension look like?

```
[  x ^ 2  |  x<-[1..10] ,  even x ]
```

expression — generator — filter

Each generator introduces a new local variable and a list of values for that variable.

The variables may be used in the expression or later on in the generators and filters.

Each filter is a conditional expression which cuts out some of the values generated.

You can have as many generators and filters as you want.

Guarded Equations

The equations in a function definition can contain guards.

A function that calculates Australian income tax:

```
> incomeTax x
>   | x <=  18200 = 0.0
>   | x <=  37000 = 0.190 * (x -  18200)
>   | x <=  87000 = 0.325 * (x -  37000) +  3572
>   | x <= 180000 = 0.370 * (x -  87000) + 19882
>   | otherwise   = 0.450 * (x - 180000) + 54232
```

Here "`otherwise`" is the default option (it is a synonym for `True`).

Basic Types in Haskell

Haskell has `Int`, `Integer`, `Float`, `Double`, `Bool`, and `Char`.

There is also a "unit" type `()` whose only inhabitant is `()`.

`Int` has limited-precision integers, while `Integer` is for arbitrary-precision.

The inhabitants of `Bool` are `False` and `True`.

Inhabitants of `Char` are `'a'`, `'b'`, etc.

Pre-Defined Structured Types

A tuple is written `(e1, e2, ..., en)`.

If `ti` is the type of `ei` then the tuple's type is `(t1, t2, ..., tn)`.

A list is written `[e1, e2, ..., en]`.

Every element must have the same type `t`, and the list's type is `[t]`.

The list is equivalent to

```
e1 : e2 : ... : []
```

that is, colon is used for the (infix) operator "cons."

Strings

In Haskell, a string is a list of elements of type `Char`.

We often use a type synonym for that type

```
type String = [Char]
```

Many useful char/string functions live in the module `Data.Char`.

```
swapCase :: String -> String
swapCase s
  = map swap s
    where
      swap c
        | isUpper c = toLower c
        | isLower c = toUpper c
        | otherwise = c
```

Function Space

Functions are first-class and therefore higher-order. They can be named via declarations, or they can be anonymous via lambda abstractions.

For example, `\x->x+1` is the same function as `succ`, defined by

```
> succ x = x+1
```

If `x` has type t and `e` has type t' then `\x->e` has type t `->` t'.

User-Defined Types

```
> data Bool = True | False
> data Colour = Red | Blue | Green
> data Tree a = Node a (Tree a) (Tree a) | Void
> data Point a = Pt a a
```

`Bool` is a (pre-defined) type constructor without arguments.

`True`, `False`, `Red` etc., are nullary data constructors.

`Bool` and `Colour` are enumerations—only nullary data constructors.

`Point` is a tuple type constructor, as it has only one data constructor. The others are called union types.

User-Defined Types

The general form is

$$\begin{array}{lcl} \texttt{data}\ T\ u_1\ \ldots u_m & = & C_1\ t_{11}\ \ldots t_{1k_1} \\ & | & \ldots \\ & | & C_n\ t_{n1}\ \ldots t_{nk_n} \end{array}$$

where T is a type constructor, the u_i are type variables, the C_i are (data) constructors, and the t_i are types, possibly having some of u_i as constituents.

Note that these data types may be polymorphic.

The same data constructor cannot be used in different user-defined types.

Data Constructors Are Functions

Data constructors are really just functions:

```
? :t Pt
Pt :: a -> a -> Point a
? :t Node
Node :: a -> Tree a -> Tree a -> Tree a
```

Functions Using User-Defined Types

```
> data Month
>    = Jan | Feb | Mar | Apr | May | Jun
>    | Jul | Aug | Sep | Oct | Nov | Dec
>      deriving (Eq, Ord, Show, Enum)
>
> data Season
>    = Summer | Autumn | Winter | Spring
>      deriving (Eq, Ord, Show, Enum)
>
> toSeason :: Month -> Season
> toSeason month
>    | month < Mar || month == Dec = Summer
>    | month < Jun                 = Autumn
>    | month < Sep                 = Winter
>    | otherwise                   = Spring
```

Type Classes

The '`deriving`' clauses ensures that `Month` and `Season` become members of certain "type classes" (more on these in a week's time).

Membership of '`Eq`' means that, for example, the test `month == Dec` is meaningful (and does what we expect).

Membership of '`Ord`' means that `month < Mar` makes sense.

Membership of '`Show`' means the system is able to print the members of the types.

Membership of '`Enum`' means that expressions like `[Feb .. Nov]` make sense (and mean what we think).

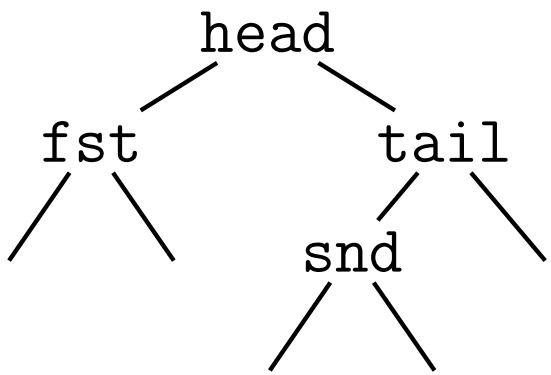
Recursively Defined Types

Many types are recursive by nature:

```
data Tree a = Node a (Tree a) (Tree a) | Void
```

One inhabitant of type `Tree String` is

```
Node "head" (Node "fst" Void Void)
            (Node "tail" (Node "snd" Void Void) Void)
```



Polymorphism

With polymorphic typing, we allow functions to be applied to a number of different possible compatible types.

```
> id x = x
>
> map f xs = [f x | x <- xs]
```

`id` has type `a -> a`.

`map` has type `(a -> b) -> [a] -> [b]`.

The types are inferred automatically.

Type Signatures

In the absence of type signatures, the most general types will be inferred:

```
? :t map
map :: (a -> b) -> [a] -> [b]
? :t id
id :: a -> a
```

It is considered good programming style to include type signatures with function definitions.

Let Us Program Bottom-Up Merge Sort

We gave quicksort as an example Haskell program above; but merge sort is a much better choice in a language like Haskell.

Here is a (recursively defined) function that merges two sorted lists, so that the result is also sorted:

```
> merge :: Ord a => [a] -> [a] -> [a]
> merge [] ys = ys
> merge xs [] = xs
> merge (x:xs) (y:ys)
>    | x <= y     = x : merge xs (y:ys)
>    | otherwise = y : merge (x:xs) ys
```

Implementing Bottom-Up Merge Sort

The next function merges consecutive pairs of lists together.

For instance `mergePass [[1,4],[2,3],[7,8],[5,6],[9]]` yields `[[1,2,3,4],[5,6,7,8],[9]]`.

```
> mergePass :: Ord a => [[a]] -> [[a]]
> mergePass [] = []
> mergePass [x] = [x]
> mergePass (xs:ys:zss) = merge xs ys : mergePass zss
```

The type signature says "`mergePass` takes a list of lists of something `a`, and returns a lists of lists of `a`, provided the type `a` is in the type class `Ord`."

Implementing Bottom-Up Merge Sort

The next function keeps applying `mergePass` on a list of lists until there is only one inner list left:

```
> mergeRepeat :: Ord a => [[a]] -> [a]
> mergeRepeat [] = []
> mergeRepeat [xs] = xs
> mergeRepeat xss = mergeRepeat (mergePass xss)
```

Now we can sort a list by making each element a singleton list and applying `mergeRepeat`:

```
> mergeSort :: Ord a => [a] -> [a]
> mergeSort xs = mergeRepeat [[x] | x <- xs]
```

The Topics for the Next Weeks

Propositional logic and mechanised reasoning.