Haskell Tutorial

José Bernardo Barros   José João Almeida

Departamento de Informática
Universidade do Minho
Braga, Portugal

September, 1998

Introduction

Haskell is a general purpose, non-strict, purely functional programming language. There are several compilers and interpreters of this language freely available for almost any computer. The language is defined in the Haskell 1.4 Report[1] and the Haskell 1.4 Library Report[2]. If you want to learn to program in Haskell, a tutorial: *A Gentle Introduction to Haskell* [1], is also available[3]. The present document should be seen as a complement to this text: it gives a hands-on tour of a small interpreter of Haskell called Hugs[4]. Throughout this text you will find small exercises that will help you getting acquainted with the language. These appear is a different font, and with a vertical bar on the left.

Hugs

The first thing that you have to do is to make sure that you have the Hugs iinterpreter, and to find out how to start it. Try the command **hugs**.

This should take you to a screen like

```
    ___    ___   ___    ___   __________   __________
   /  /   /  /  /  /   /  /  /  _______/  /  _______/         Hugs 1.4
  /  /___/  /  /  /   /  /  /  / _____   /  /______
 /  ____   /  /  /   /  /  /  / /_   /  /______   /  The Nottingham and Yale
/  /   /  /  /  /___/  /  /  /___/  /  _______/  /    Haskell User's System
/__/   /__/  /_________/  /_________/  /_________/          June 1998

  Copyright (c) The University of Nottingham and Yale University, 1994-1998.
```

[1] URL: `http://www.haskell.org/report/index.html`
[2] URL: `http://www.haskell.org/library/index.html`
[3] URL: `http://www.haskell.org/tutorial/index.html`
[4] URL: `http://haskell.org/hugs/`

```
    Bug reports: hugs-bugs@haskell.org.   Web: http://www.haskell.org/hugs.

Reading file "/usr/local/share/hugs/lib/Prelude.hs":

Hugs session for:
/usr/local/share/hugs/lib/Prelude.hs
Type :? for help
Prelude>
```

The last line points out that HUGS is ready to execute commands. There are several commands that you may want to execute:

:quit will end the session

:? will list the commands available

Apart from these commands, HUGS can compute the value of expressions. Thus, if you type in `3+4`, it will answer back `7`.

Types

Haskell is a typed language. This means that each expression (or term) has a type. You can ask the type of an expression using the command `:type`; we'll take a look at this later. But you can also instruct HUGS to print the type of each computed result, by entering the command `:set +t`. Try this command and then compute some basic arithmetics.

The basic types in HASKELL are:

- `Int` and `Integer` are used to represent integers. Elements of `Integer` are *unbounded* integers.
- `Float` and `Double` are used to represent floating point numbers. Elements of `Double` have higher precision.
- `Bool` is the type of booleans: `True` and `False`.
- `Char` is the type of characters.

Notice that all the names of types start with a capital letter.

Apart from these basic types, there are several ways of making new types:

- if `a` is a type, `[a]` is the type of the sequences of elements of `a`
  - `[]` is the empty sequence
  - `h:t` is the sequence whose head and tail are `h` and `t` respectively

  The sequence with the first three natural numbers is thus represented by

```
0:1:2:[]
```

Alternatively, we can simply write `[0,1,2]` to represent that sequence.

The particular case `[Char]` has another name – `String` and there is another way of representing these sequences: by delimiting them by quotes. Thus, the sequence

```
'H':'a':'s':'k':'e':'l':'l':[]
```

can also be written as

- `['H','a','s','k','e','l','l']`
- `"Haskell"`

- if `a` and `b` are types, `(a,b)` is the type of pairs whose first component is of type `a` and second component is of type `b`. Of course, this construction may be done for more than two types `a` and `b`.
- if `a` and `b` are types, `a -> b` is the type of functions from `a` to `b`

1. Find expressions whose type is
   - `(Bool,[Char])`
   - `([Bool],Char)`
   - `[(Bool,Char)]`

   Test your answers by using HUGS to evaluate the types of those expressions.
2. Using the command `:type`, find the type of the following expressions
   - `head`
   - `sum`
   - `fst`
   - `elem`
   - `flip`
   - `flip elem`

By supplying the expected arguments to the above functions, try to guess what they are.

There exist a lot of functions that are readily available when you start HUGS. Their definitions are stored in a file called `Prelude.hs`. That is the reason for the line

```
Reading file "/usr/local/share/hugs/lib/Prelude.hs":
```

We can also have our own definitions. These should be written in a file and then loaded using the command `:load`. It is usual to name these files with a postfix `.hs` (for haskell script).

Using your favourite text editor, create a file named `example.hs` with the following definitions

```
square x = x * x

factorial x = product [1..x]
```

Load this file into HUGS, by typing `:load example` (HUGS will assume that the file ends with `.hs`). You can now use the definitions of the functions `square` and `factorial`. Test these definitions by evaluating the following expressions:

- `square 4`
- `square (factorial 3)`
- `factorial (square 3)`

You might have noticed by now that HUGS *"guesses"* the types of the expressions that you ask it to evaluate. But you can also provide this type with the expression.

After instructing HUGS to print the types of the expressions (by using the command `:set +t`), evaluate the following:

- `3 + 4`
- `(3::Integer) + 4`
- `factorial 50`
- `factorial (50::Integer)`

Similarly, you can provide type information in your scripts:

Edit the file `example.hs` in order to obtain:

```
square :: Float -> Float
square x = x * x

factorial :: Integer -> Integer
factorial x = product [1..x]
```

Reload the file (using the command `:reload`) and re-evaluate the expressions above.

There exist a lot of functions to manipulate lists. You can find out the complete list by consulting the on-line guide that comes with HUGS[5].

[5] file://usr/local/share/hugs/docs/library/index.html

1. Define functions to:
   (a) compute the length of a list
   (b) compute the concatenation of two lists
   (c) reverse of a list
   (d) merge two sorted lists
   (e) sort a list (for instance, using quicksort)
2. Define a function `squares` that computes the list of squares from a list.

One very important feature of most functional programming languages is the possibility of defining functions that receive other functions as arguments. For instance, the function `filter` can be defined as

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (h:t) = let t' = filter p t
                 in if (p h) then h:t'
                              else t'
```

1. The function `map` has type:

   ```
   map ::  (a -> b) -> [a] -> [b]
   ```

   `map f l` is the list that results from applying the function `f` to each element of `l`. Define it.
2. Use the function `map` to define `squares`.
3. The function `foldr` has type

   ```
   foldr ::  (a -> b -> b) -> b -> [a] -> b
   ```

   and `foldr f e [a,b,c] = f (a, f (b, f (c,e)))` Define `foldr`.
4. Use `foldr` to define the functions `length`, and `sum`.

Inductive Types

HASKELL also provides a way of defining inductive types. These start with the keyword `data`.

Enumerated Types

The simplest inductive types that one can define are enumerated types. For instance, to define a type for the days of the weak, one can use such a construction:

```
data WeekDays = Sunday | Monday | Tuesday | Wednesday |
                Thursday | Friday | Saturday
```

This declaration defines a new type – WeekDays with seven elements. These elements are called *data constructors* or simply *constructors*.

Note again that the name of the type starts with a capital letter. Moreover, the names of the constructors must also start with a capital letter.

This example takes us to a peculiarity of HASKELL – the meaning of the layout of a program. In the majority of programming languages, definitions have delimiters that point out where that definition starts and ends. In HASKELL this effect is obtained with a specific layout. Formally, a definition ends before the first piece of text which lies at the same level or to the left of the start of the definition. Thus, the following text

```
a b c
   d e
     f
     g
   h
i j k
  l m
   n
```

should be seen structured as

```
a b (c (d e f g) h)
i j k l m n
```

Let us now write a function that takes an element of WeekDay and returns whether it is a working day. We will define this function by providing an equation for each of the possibilities of elements of that type:

```
workingDay :: WeekDay -> Bool
workingDay Sunday    = False
workingDay Monday    = True
workingDay Tuesday   = True
workingDay Wednesday = True
workingDay Thursday  = True
workingDay Friday    = True
workingDay Saturday  = False
```

Note that as the patterns used are non-overlapping, the order in which they appear in the program is irrelevant.

HASKELL also allows the use of overlapping equations. In that case one should be careful with the order in which these equations appear. When two equations can be applied to the same expression, the one chosen is the one which appears first in the program.

Thus, the previous definition could also be written as

```
workingDay :: WeekDay -> Bool
workingDay Sunday    = False
workingDay Saturday  = False
workingDay x         = True
```

> Define a function that, given a working day, returns the following day.

Recursive Types

Recursive types can be defined using induction. For instance, the natural numbers can be defined by:

```
data Nat = Z | S Nat
```

Again, this declaration defines a new type – `Nat`. Associated with this new type, there exist two (data) constructors:

- `Z` is an element of `Nat`
- `S` is a function that given an element of type `Nat`, yields a (new) element of type `Nat`

Thus, `Z`, `S Z`, or `S S S S Z` are all elements of type `Nat`.

Let us define a function that takes an element of type `Nat` and returns whether that element is zero.

This function is defined by *pattern-matching*:

```
isZero :: Nat -> Bool
isZero (S x) = False
isZero Z = True
```

> 1. Define a function `toInt` that converts a natural number into an integer.
> 2. Define a (recursive) function `oddN` that tests whether a natural number is odd.
> 3. Redefine the function `oddN` using `toInt` and `odd`.

Parametric Types

Inductive types can be used to define parametric types.

For a given type `a`, the type `Maybe a` is defined as

```
data Maybe a = Nothing | Just a
```

Note that `Maybe` **is not** a type – it is a *type constructor*, for it takes a type and yields a type.

The type `Maybe a` can be used to represent the result of a partial function.

> Using pattern matching, define a function that *adds* two elements of type `Maybe Int`.

An example of a recursive and parametric type is that of binary trees whose nodes are of some type `a`:

```
data BinTree a = Null | T a (BinTree a) (BinTree a)
```

> 1. Define the function `inorder :: BinTree a -> [a]` that returns the list of elements of a tree.
> 2. Using pattern matching, define a recursive function that sums the nodes of a binary tree of integers.
> 3. Redefine the previous function so that it can be used to *add* the elements of a binary tree of `Maybe Ints`.
> 4. Similarly to what happens with the function `foldr`, define a higher order function `foldBtree` that can be used in the definition of the two functions above.

> In order to simulate a change giving machine (in PTEs) we will use the type `Coins` and the list `values` defined as.
>
> ```
> type Coins = [Int]
> values     = [200,100, 50, 20, 10,  5,  1]
> ```
>
> Each element of type `Coins` will represent the number of each of the coins available. Thus `[1,0,2,0,0,1,3]` means 1 coin of 200 PTEs, 2 coins of 50, 1 coin of 5, and 3 coins of 1.
>
> 1. Define the functions that *add* and *subtract* two elements of type `Coins`.
> 2. Define the function `amount :: Coins -> Int` that computes the amount of money corresponding to a set of coins.
> 3. Define a function `payment :: Coins -> Integer -> Maybe Coins` that simulates the payment of a certain amount using a particular set of coins. The result is the set of coins used (the fact that it is `Maybe Coins` explicits the fact that the payment may not be possible).

> The fact that HASKELL is a lazy language, allows us to define infinite structures. For instance, `[0..]` represents the list of all natural numbers, whereas `[x | x <- [0..], odd x]` represents the list of all odd numbers. Define a function that computes all prime numbers.

Classes

One way to understand classes in Haskell is to view them as *types of types*. Another possible approach is to talk about classes as a means of expressing (ad-hoc) polymorphism.

> Use Hugs to compute the following expressions (make sure that Hugs prints out the type of the computed expressions):
>
> - `3 + 4`
> - `3.0 + 4.0`
> - `(3::Integer) + 4`

What is then the type of the function `+`? After all, it can be used to

- add two `Int`s yielding an `Int`,
- add two `Double`s yielding a `Double`,
- add two `Integer`s yielding an `Integer`,

But you cannot compute `'a'+'b'`.

One way to solve this problem is to *group* types into **classes**, in the same way that expressions were *grouped* into types.

When asking Hugs for the type of `+` we get the following answer:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude>
```

This answer should be read as *`+` is a function that takes two elements of a type `a` and returns an element of the same type `a`, for every type `a` which is an instance of the class `Num`.*

The declaration of a class in Haskell is done using the keyword `class`, and by enumerating all the functions that should be available for the instances of that class.

For instance, one of the simplest (and more used) classes in Haskell is the class `Eq`, defined as

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

This definition should be read as

> For a type `a` to be an instance of class `Eq` there must exist functions

```
== ::  a -> a -> Bool
/= ::  a -> a -> Bool
```

In order to state that a particular type is an instance of the class `Eq`, one needs to explicit the way in which elements of that type are compared. For instance, to declare that the type `Maybe Int` is an instance of the class `Eq`, one might type the following:

```
instance Eq (Maybe Int) where
    Nothing == Nothing   = True
    (Just x) == (Just y) = (x == y)
    _ == _               = False

    x /= y = not (x == y)
```

Note that, in the third line of this definition, there are two occurrences of `==`

- the first refers to the function that we are defining – comparison of elements of type `Maybe Int`
- the second refers to the comparison of elements of type `Int` (which is an instance of this same class!).

This definition would *work* not only for the type `Maybe Int` but for any type `Maybe a`, provided that the type `a` is itself an instance of `Eq`. We can say this with the following definition:

```
instance (Eq a) => Eq (Maybe a) where
    Nothing == Nothing   = True
    (Just x) == (Just y) = (x == y)
    _ == _               = False

    x /= y = not (x == y)
```

Consider the following definition

```
class Set s where
   empty     :: s a
   isEmpty   :: s a -> Bool
   singleton :: a -> s a
   union     :: s a -> s a -> s a
   member    :: a -> s a -> Bool
   choice    :: s a -> (a , s a)
```

1. Lists can be used as sets.

   ```
   data SetsasLists a = SL [a]
   ```

   Complete the following definition

   ```
   instance Set SetasLists where
      .............
   ```

2. Lists without duplicates can also be seen as sets. How would you change the previous definitions to define this instance?

3. Complete the following definition:

   ```
   instance (Eq a) => Eq (SetasLists a) where
      .............
   ```

Monads

The class `Monad` is defined in HASKELL as

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (b -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   x >> y = x >>= (\ a -> y)
```

Note that this is a constructor class (as opposed to a type class like `Eq`) – its instances are type constructors.

The operation `>>=` is usually called **bind**.

One way to understand the use of monads in functional programming is to see an expression of type `M a` (for some monadic type constructor `M`) represents a computation of type `a`. Under this point of view, the operations available can be interpreted as

- `return x` represents a computation whose result is `x`
- given a computation `c` (of type `a`) and a function `f` that takes an element of type `a` and performs a computation of type `b`, the expression

```
c >>= f
```

is the computation that starts by performing computation `c`, and then performs the computation `f x` where `x` is the result of the computation `c`.

- the operation `>>` is similar to the previous one, except that the intermediate value (`x`) is ignored.

Let's start with the simplest way to represent a computation:

```
data Id a = a   -- This is not valid Haskell code
instance Monad Id where
    return x = x
    x >>= f  = f x
```

This corresponds to the *classical* view of computations in functional programming – executing a computation corresponds to the evaluation of a normal form.

In the next case, a computation may yield a value of a certain type `a`, or give no result at all. An appropriate type for this is the type `Maybe a` defined above. The definition of `Maybe` as a monadic constructor is as follows:

```
instance Monad Maybe where
   -- return :: a -> Maybe a
   return a = Just a
   -- (>>=) :: (Maybe a) -> (a -> (Maybe b)) -> Maybe b
   Nothing  >>= _ = Nothing
   (Just x) >>= f = f x
```

The natural generalization to this example is to think of non-deterministic computations – that can yield a finite number of results. Lists are a good candidate for this type. The list constructor can be seen as monadic with the following definitions:

```
instance Monad [] where  -- This is not valid Haskell
   -- return :: a -> [a]
   return x = [x]
   (>>=) :: [a] -> (a -> [b]) -> [b]
   l  >>= f = concat (map f l)
```

There is in Haskell a syntactic alternative to the use of the operators `>>=` and `>>`. This alternative is inspired in the definition of lists by compreenshion.

Instead of writing something of the form:

```
c1 >>= ( \ x ->
c2 >>
c3 >>= ( \ z -> f)))
```

one can write

```
do {  x <- c1 ;
      c2 ;
      z <- c3 ;
      f
   }
```

One final example is that of computations with an internal state. This can be achieved by using state transition systems

```
data StTransf state value  = T (state -> (value, state))
```

The constructor `StTransf state` can be defined as an instance of the class `Monad`:

```
instance Monad (StTransf state)
    where return a = T (\x -> (a,x))
          ((T f) >>= g)
             = T (\s -> let (a,s') = f s
                            T fun  = g a
                        in  fun s')
```

Let us now use this monad in a very simple way – the state will only keep track of the number of additions made.

```
-- Basic operations
add  a b  = T (\s -> ((a+b), s+1))
sub  a b  = add a (-b)
mult 0 b  = return 0
mult (n+1) b = do { x <- mult n b ;
                    add x b
                  }
-- interrogations
state = T (\s -> (s, s))
resetstate = T (\s -> ((),0))
```

The use of these basic operations is very simple and resembles an imperative program:

```
prog1 = do { x <- add 3 4 ;
             y <- mult 3 x ;
             z <- add x y
             z <- add z 1 ;
             return z
           }
```

The type of this program can be checked using Hugs:

```
Main> :t prog1
prog1 :: StTransf Int Int
```

To execute this program, we have to provide an initial state:

```
execute (T program) = program 0
```

Let's test the behaviour of `prog1`

```
Main> execute prog1
(29,6)
```

Meaning that the returned value is 29 and that the final state is 6.

> Change the above example so that the state will keep also track of the smallest computed number.

An important application of monads in HASKELL is the input/output. In this viewpoint, an interactive program is just a computation that may perform some I/O. The type `IO a` pre-defined in HASKELL, reflects this idea – an element of this type is a program that performs some I/O and returns a value of type `a`. The type constructor `IO` may be defined as an instance of `Monad`:

- `return x` is the computation that performs no I/O at all and returns the value `x`
- `p1 >>= f` is the program that starts by performing the `p1`'s I/O and then performs the I/O correspondent to `f x`, where `x` is the value returned by `p1`. In a certain way, this operation corresponds to the sequential composition of interactive programs.

> The following are pre-defined functions in HASKELL:
>
> - `putChar :: Char -> IO ()`
> - `getChar :: IO Char`
>
> Define the following (pre-defined) functions:
>
> 1. `putStrLn :: String -> IO ()`
> 2. `getLine :: IO String`

References

[1] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. Technical report, Department of Computer Science, Yale University, 1992.