Tips on Haskell

Nik Sultana*
Mathematical Institute
University of Munich

August 17, 2008

1 Introduction

This guide seeks to emulate the style used in Tofte's summary [6] of Standard ML. It seeks to outline some essential pieces of syntax one needs to know in order to use Haskell, and loosely describes their semantics.

The coarse nature of this guide makes it better suited for readers familiar with the functional paradigm who want a quick outline of the language, or who want to do a rapid revision of Haskell syntax. Thompson's book [5] on Haskell is recommended for more expansive coverage, including methods of proof for Haskell programs.

2 Practical matters

The latest version of the Haskell language is *Haskell 98* and its last minor revision was made in 2003 [3]. This language has been implemented by various groups to produce compilers and interpreters. At present *GHC*[1] is the most widely-used compiler for Haskell. It also implements a host of extensions to Haskell which, although some of them are experimental, often serve to express clearer programs.

When one is getting to grips with a language an interpreter can be more useful than a compiler. GHC is distributed with an interpreter called *GHCi*. The interpreter *Hugs*[2] is also quite popular. Both GHC and Hugs can be run on various platforms. The code presented in this guide has been checked using GHCi version 6.4.1.

Contents

*nik.sultana@yahoo.com

[1] `http://www.haskell.org/ghc`

[2] `http://www.haskell.org/hugs`

2.1 Command line matters

Haskell source files can be recognised from their ".hs" extension.

The interpreters can be started by typing `hugs` or `ghci`; these may optionally be followed by the name of the Haskell file to be loaded. Typing `:?` in the top-level will describe directives recognised by the interpreter. The behaviour of either interpreter can be tuned by means of switches at the command line, but only "standard" operation will be needed in the examples that follow.

In order to compile a source file, say `Source.hs`, into an executable named `go` use `ghc -c Source.hs -o go`. Multiple source files may be provided. Typing `ghc --help` provides further details on how the compiler's behaviour can be controlled.

3 Functional paradigm

This section is intended to reinforce the mode of thinking underlying functional programming. This will be approached using a comparison with imperative programming.

Programming in the imperative style consists in making explicit the sequence of steps needed to derive output from input. Frequently in imperative programs one gets work done by explicitly directing the computer to move values around memory and transform the contents of memory locations.

In functional programming the relation between input and output is described in a more direct manner: rather than moving values around memory storage and manipulating the contents of memory cells, the values are manipulated directly. This sustains an abstraction: in functional programming one is describing mathematical objects rather than prescribing operations to a machine. The following section will describe characteristics through which Haskell fulfils the expectation that "the essence of functional programming is expression evaluation" [4, p.3].

4 Semantics

Haskell is described as being a *lazy* and *purely-functional* language. 'Lazy' means that expressions are evaluated only if their values are needed, and once evaluated their corresponding values are stored to avoid recomputing them in the future. 'Purely-functional' means that Haskell functions behave like normal mathematical functions – that is, when applied to the same arguments they always evaluate to the same value. This contrasts with an imperative input command, for example, which may potentially return a different result each time it is called. A way of defining so-called "impure" functions in Haskell will be described in §15.

Haskell is also a *statically typed* language: all values pertain to types, and types are inferable at compile-time.

5 Comments

Single-line comments are preceded by `--` and extend to the end of the line. Multiline comments are enclosed between `{-` and `-}`.

6 Prelude

A kernel of oft-used definitions in Haskell are contained in its so-called *prelude*. Other useful definitions can be found in the standard libraries accompanying Haskell. As with any other language, familiarity with the prelude and libraries pays off in speed and elegance in the programs one writes. Haskell programmers are also fortunate to have search engines for both preludes and libraries. One of these engines is called Hoogle[3] and has been around for a few years. Searching can be done using not only definition names but also type signatures.

Some of the definitions given in this guide are available in the prelude or in some library, and need not be redefined in practice. Standard definitions will appear underlined in code fragments.

7 Definitions

This section concerns solely the definition of *values* (including functions); other forms of definitions – such as types and modules – will be described later.

Values in Haskell are named and defined in the form of *defining equations*, taking the following form:

```
f x0..xn = expr
```

Here `f` is an identifier chosen to be the function's name, `x0..xn` is a (potentially empty) space-separated sequence of identifiers that name its formal parameters, and `expr` is an expression. The expression need not directly follow the symbol `=` and may start on a different line, as long as the the first symbol of `expr` is below and to the right of the first symbol of `f`. This is known as the *offside rule*.

[3] Accessible at `http://www.haskell.org/hoogle/`

The following simple examples are all legitimate definitions:

```
five      = 5
add   x y = x + y
undef     = undef
```

The last example shows the smallest recursive definition; defining equations in Haskell are in fact *recursion equations* and Haskell's semantics guarantee solutions to these equations. The solution to the third equation is a value that is *undefined*, usually denoted by the symbol $\perp$.

The construction of expressions will not be addressed directly through grammar definitions here, but conveyed through examples. The first example shows a conditional expression used to define the factorial function – well-loved by functional programmers:

```
fact n = if n == 0
         then 1
         else n * (fact $ n - 1)
```

Note that the symbol = in Haskell is reserved for definitions, and that the equality test is denoted by the == symbol. The symbol $ denotes function application; this is normally denoted by juxtaposition, but making it explicit in this case saves on brackets.

Individual definitions may be given using multiple equations to improve readability. For example, the Fibonacci numbers are the solution of the recurrence relation $F_n = F_{n-1} + F_{n-2}$ under seed values $F_0 = 0$ and $F_1 = 1$. This may be expressed in Haskell as follows:

```
fib 0 = 0
fib 1 = 1
fib n = (fib $ n-1) + (fib $ n-2)
```

Instead of using multiple equations one could use `case` expressions, demonstrated next.

```
fib n = case n of
          0 -> 0
          1 -> 1
          n -> (fib $ n-1) + (fib $ n-2)
```

Alternatively one could structure definitions using *guards* – a notation in which the conditions affecting the behaviour of a function are given using Boolean combinations:

```
fib n
    | n==0      = 0
    | n==1      = 1
    | otherwise =
       (fib $ n-1) + (fib $ n-2)
```

Guards make explicit the tacit Boolean tests that the definition systems described previously perform: the behaviour of the latter is predicated on *pattern matching* which occurs at the equation or case level. Patterns are syntactical objects which may contain identifiers. If the input to a function matches the pattern, then free identifiers within the pattern are bound in the expression that forms the function's definition. So far we have only seen patterns over variables ranging over integers, but "structured", or algebraic types, will enrich the language of patterns. This will be covered in §12.

An expression may also contain *local declarations* through use of `where` and `let` expressions. The following definition of `double` has the definition of `twice` local to its scope: no other definition in the program can "see" this definition.

```
double n = twice (+n) 0
           where twice f = f . f
```

This definition also serves as an example of *partial application*: the function + is applied to a single argument thus yielding a new function that takes a single argument (the remaining argument that was to be given to +). Binary infix operators can be defined by enclosing the operator's symbol in round brackets. The following definition duplicates the addition operator:

```
(%%) x1 x2 = x1 + x2
```

Definitions may be complemented by a *type signature*; this goes a little way towards specifying the behaviour of the function being defined. The language of type signatures will be described in the rest of this guide, but for now it suffices to say that `Integer` is the type of integers, `Int` is a limited-precision form of the type `Integer`, and the binary `->` forms the type of functions.

```
nSign :: Integer -> Integer -> Integer
nSign n x =
      let signum :: Integer -> Integer
          signum 0 = 0
          signum n = if n < 0
                     then -1
                     else 1
      in n * (signum x)
```

The previous definition also serves as an example of using `let` for local definitions.

8 Types

Before proceeding it would be useful to expand on what has been said about types. Values can be either *observable* in which case they are said to be of

ground type, or else they are *non-observable* due to being, or containing, general mathematical functions. In Haskell practice, however, values of a type are observable if that type is shown to belong to a class of "showable" types; this will be expanded further in §13.1.

The grammar of types has basic (ground) types as terminal symbols. Some examples of such types are given below, together with examples of inhabitants of that type. Note that these types are *native* to Haskell.

Type	Values
`Bool`	`True, False`
`()`[4]	`()`
`Integer`	`...-1,0,1,...`
`Char`	`'','a','b',...`
`String`	`"","a","aa",...,"a","ab",...`

The core mechanism for constructing other types involves using function types, however other methods can be used to facilitate this. For instance, n-tuple types can be defined explicitly as a tuple of types and will have values drawn from the (lifted) Cartesian product of those types. A general way of building types will be described in §12.

Frequently-used composite types can be given a name instead of being explicitly defined in each place they are used. This involves defining a *type synonym*. For example, the following defines a name to stand for the product of Booleans and integers. Note that type names must start with an uppercase alphabetical symbol.

```
type BInt = (Bool, Integer)
```

Rather than define a synonym it might be preferable to distinguish the new type from the composition of its parts. This distinction will result in increased abstraction of the types being composed. This kind of type definition is sketched below.

```
newtype T' = Constr T
```

Note that `Constr` stands for a *constructor*: it can be regarded as an n-ary function that maps values from its argument-types into some value in the type being defined. Note that a constructor is named using an identifier that starts with an uppercase alphabetical symbol.

Finally, type variables may appear in type signatures in Haskell since a restricted form of polymorphism is allowed. Such variables are denoted by identifiers consisting of lowercase alphabetical symbols, conventionally starting with `a`. Parametric polymorphism in Haskell will be described further in §10.

[4]This symbol denotes Haskell's unit type. Note that it also denotes its sole inhabitant; any confusion is clarified by the context in which the symbol occurs.

Type variables may also appear in the definition of other types – usually indicating that the type being defined is a *container* type. This will be described further in §9.1, §11, and §12. For example, the type `[a]` is the type of lists of type `a` – that is, any other type.

Haskell also supports overloading of functions, or 'ad hoc' polymorphism, through a mechanism called *type classes*. The type signatures of overloaded functions contain type variables which are restricted by a context of a particular type class. This will be described in §13.

9 Functions

Apart from named functions through definitions, one can express *anonymous functions* using abstraction-like notation from the λ-calculus. Let `expr` be an expression of type `T`, then `\x-> expr` is another expression denoting a function of type `T' -> T`. The type represented by `T'` is inferrable.

Let definition `f` be of type `T' -> T''` and `g` of type `T -> T'`, then the *composition* of the two functions is expressed using `f . g` and is typed `T -> T''`.

A function having a single argument consisting of an n-tuple can be turned into a function having n arguments; this is called *currying*. The reverse direction is also possible in general.

9.1 Functionals

Since functions are first-class citizens in Haskell's world they may be passed as arguments – and returned as values. A functional is defined next, then used to restate the factorial function defined in §7.

```
fixp :: (a -> a) -> a
fixp f = f (fixp f)


fact' f n = if (n == 0)
            then 1
            else n * (f $ n - 1)

fact = fixp fact'
```

10 Parametric polymorphism

Some functions behave uniformly irrespective of the type of values over which they are defined. A restricted, though practically very useful, class of these functions can be described directly within Haskell. A very simple example of these functions is the identity function, defined next.

```
id :: a -> a
id x = x
```

The type of the polymorphic functional `map` is described next; its definition will be given in the next section about the type of lists.

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

11 Lists

Lists have a special place in functional programming, both historically and practically. They are *container* types since they principally serve as structures that can contain values of other types.

Lists consist of values constructed in two ways: using a nullary *nil*, denoted by `[]`; alternatively using a binary *cons*, denoted by infix `:`, of which first argument consists of a value and the second argument consists of the remainder of the list – called head and tail respectively. The following example shows two examples of lists of type `[Integer]`; they encode the same list and the second shows a more pleasant notation that could be used.

```
x = 5:3:1:[]
y = [5,3,1]
```

Lists can also be defined by using abbreviations for arithmetic series as in the next two examples. The third definition uses *comprehensions*.

- `nats = [1..]`
- `odds = [1,3..]`
- ```
  compound = [(x, y, x+y)
              | (x,y) <- (zip nats odds)]
  ```

When defining functions over lists pattern matching proves to be an invaluable tool. Furthermore, since lists may be analysed and built without examining the values they contain, one often finds functions defined over lists to be polymorphic. The next definition is an oft-quoted example.

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + (length xs)
```

The function `map`, the type signature of which was given in the previous section, is defined next. Its definition shows that given a function and a list, it applies the function pointwise to the list.

```
map f []     = []
map f (x:xs) = (x':rest)
               where x'   = f x
                     rest = map f xs
```

12 Algebraic types

This provides a general mechanism for building new types rather directly by describing their inhabitants. Note that $\perp$ tacitly inhabits every Haskell type. A simple example of using this device involves defining an enumerated type consisting of three elements as shown next.

```
data Answer = Yes | No | Unknown
              deriving (Eq,Show,Read)
```

The keyword `deriving` automatically instantiates the type being defined in a restricted set of type classes – such as those for values for which equality is decidable, and readable and showable values.

Type definitions may also be recursive, and may themselves be parametrised by other types. The following example defines binary trees and is followed by a polymorphic function over trees. Note that the symbol _ in the definition of `height` is a dummy identifier used to discard values that would otherwise have been bound to an identifier standing in place of the _ symbol. The operator `@`, read *as*, serves to name a pattern-matched input argument. In this example `inp` is not used and serves only to demonstrate use of the `@` symbol. In Haskell any binary function can be applied infix by enclosing it in backticks, as with `max` in the definition of `height`.

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

height :: (Tree a) -> Int
height Leaf = 0
height inp@(Node _ t1 t2)
           = 1 + height t1 `max` height t2
```

The datatype of trees can also be defined using *records* defining projection functions from tree values. The definition of `height` need not be changed to accommodate the modification to `Tree a` shown below.

```
data Tree a = Leaf
            | Node {val :: a, left :: Tree a,
                    right :: Tree a}
```

Often extensions to Haskell concern extensions to its type system. These usually facilitate the encoding of types which would otherwise be unwieldy to encode using algebraic types – and possibly impossible to encode using algebraic types alone.

12.1 Impact of laziness

As a result of lazy evaluation one can encode infinite and partially-defined objects in Haskell, knowing that
```

they will only be evaluated to the (hopefully finite) extent required by the program.

- `n3 = [1,2,3]`
- `ones = 1:ones`
-
```
aList = [1,2,3,undef]
        where undef = undef
```

Each of the previous definitions inhabit the type of lists of integers but the nature of each list is very different from the others: the first list is a finite and fully-defined; the second is an infinite list of 1's; the third is a finite but partially-defined list.

Returning to the example of Fibonacci numbers, their definition will be turned into a list next. This definition uses the definition `fib` given earlier in §7. Note that `nat` defines the list of natural numbers, and `fibs` applies `fib` to `nat` pointwise.

```
nat = 0:(map (+1) nat)
fibs = map fib nat
```

In order to project the nth element from a list the infix operator `!!` is used. It would not be difficult to prove that, for any n, `fib` n evaluates to the same value as `fibs!!`n.

The Haskell definitions of Fibonacci numbers given so far are inefficient since their evaluation involves recomputing previously-computed values: they closely follow the specification of the sequence in §7 too closely. Recomputation can be spared by storing overlapping values used in the computation of F_{n-1} and F_{n-2}, called *memoisation*. The following definition has improved complexity by means of this device.

```
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

The previous definition relies on the standard function `zipWith`; it is reproduced below from the standard prelude.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

13 'Ad hoc' polymorphism

Apart from functions that behave uniformly over different types – encountered in §10 – we often come across functions that behave differently over different types, but all their different behaviours are conceptually similar and we may want to emphasise this by using the same function symbol defined over different types. The function is said to be *overloaded*.

For instance, it is convenient to have the symbol `+` associated with an operation over all numeric types, but it is inevitable that the function it performs differs according to the type – adding reals is very different from adding naturals.

A class of types is defined by specifying functions – called *methods* – which must be applicable to all members of the class – though which they are free to implement in different ways. This is reminiscent of the signature of an algebra.

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

The code above is drawn from Haskell's prelude and describes the methods that a type must implement in order to gain membership in the numeric class. The type of natural numbers is defined next as an algebraic type then instantiated in the `Num a` class.

```
data Nat = Z | S Nat
           deriving (Eq,Show,Read)

instance Num Nat where
 Z + n         = n
 (S n) + m     = S (n + m)
 Z - _         = Z
 n - Z         = n
 (S n) - (S m) = n - m
 Z * _         = Z
 (S n) * m     = m + (n * m)
 negate x      = x
 abs x         = x
 signum x      = x
 fromInteger 0 = Z
 fromInteger n =
    if (n > 0)
    then S (fromInteger $ n-1)
    else error "Cannot cast negative integer
                into Nat."
```

The function `error` used in the previous definition has the following type:

```
error :: String -> a
```

Instead of causing the program to abort it might be preferable to use a mechanism to raise and capture exceptions. This can be done through monads – to be outlined in §15 – or alternatively "partial values" can be used: one can allow the option of not returning a value. This can be done by "wrapping" the return values in values of the `Maybe a` type, which is defined next and is part of Haskell's Prelude.

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

13.1 Readable and Showable values

Values which at some point need to be input or output need to be changed to or from values of `String` type. The classes `Show a` and `Read a` collect types that are printable and readable, respectively. For types to belong to these classes they need to instantiate the methods `show` and `read` respectively; their types are described below. Note that in these signatures the operator `=>` seeks to restrict the range of type variables to particular classes – in this case instances of `a` are being restricted to members of classes `Show` and `Read` respectively.

```
show :: (Show a) => a -> String
read :: (Read a) => String -> a
```

14 Modules

Haskell's module system is remarkably simple. Through modules one defines scopes which may be extended to include other scopes upon importing the module.

Module names start with an uppercase alphabetical symbol, and modules are defined as in the following sketch. Note that there is no need to mark the end of the module – it ends when the containing file ends.

```
module MName (exportList) where

import AnotherModule
import qualified OtherModule
```

The module's interface is defined by its *export list*: a comma-separated list of type and value definition names that are introduced to the importing scope. If the (meta) phrase `(exportList)` is omitted then everything contained in the module is exported. Modules themselves may import other modules – as long as cycles are not created – and when importing a module one can choose to limit its interface by *hiding* certain parts of its export list – this is usually done to avoid name clash resulting from a definition in the export list sharing the name of another definition already in scope. If both these definitions are required then one can specify the names of imported definitions to be *fully-qualified* – indicated by the modifier `qualified` in the code sketch above.

There are some more module-related rules:

- A Haskell file may only contain a single module.
- The name of a module must match the name of the file containing it.
- Modules cannot themselves contain other modules.

A runnable example of a module will be given next.

14.1 Abstract Datatypes

By being selective of what to export from a module we can hide the concrete types of some definitions; by doing so we can define abstract datatypes. A stack ADT is defined next.

```
newtype T' = Constr T

module Stack (
 Stack,
 newStack,
 push,
 pop,
 isEmpty
) where

newtype Stack a = Stk [a]

newStack = Stk []

push (Stk s) v = Stk (v:s)

pop (Stk (v:s)) = (v, Stk s)

isEmpty (Stk s) = case s of
                    [] -> True
                    _  -> False

instance Eq a => Eq (Stack a) where
 (Stk x) == (Stk y) = x == y
```

From the script one can observe that the concrete type of the stack is a list. This is not inferable from outside the module despite exporting the type `Stack` since the constructor `Stk` is not being exported. This hidden information shields the concrete type information from external view, making it necessary to use solely interface functions in order to manipulate stacks.

The end of the script shows an instantiation of stacks in the class of types of which values may be compared for equality. This property depends on whether we can contain the objects within the stack for equality; this dependency is reflected in the specification of the instantiation block.

15 Monads

Monads have the reputation of being one of Haskell's most obscure features. It has been said that this is in

Input	Output
`putChar :: Char -> IO ()`	`getChar :: IO Char`
`putStr :: String -> IO ()`	
`putStrLn :: String -> IO ()`	`getLine :: IO String`

Table 1: Standard functions for string I/O

Input	Output
`print :: (Show a) => a -> IO ()`	`readLn :: (Read a) => IO a`
	`readIO :: (Read a) => String -> IO a`

Table 2: Standard functions for "generic" I/O

large part due to their obscure name. They originate in category theory and, roughly-speaking, serve to interface the pure world of functions with solutions to real-world problems. Gordon [2] elaborates on different approaches to this problem.

The purpose of monads is to make the sequence of computation explicit. In effect, this distinguishes functions that occur at different stages in the process of deriving output from input: that is, calling the same function from different points in this process might yield a different value.

In the abstract, monads can be described by the following class definition.

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

The operation `>>=` (read *bind*) sequences computations into a process, and `return` introduces values to form a process. Monads stratify functions; a monad has been described as a "sticky" tag on types which a function cannot lose if it contains a monadic part. For instance, a function that at some point uses the I/O monad reflects this in its type signature. This is not generally true, however, since Haskell provides a backdoor, by means of the function `unsafePerformIO`. This is described further in §16.1.

Monadic laws will be described next in order to specify the expected behaviour of these operations. In §13 it was suggested that type classes are signatures of Σ-algebras; the following equations correspond to the laws that the Σ-algebras are expected to respect. Here ≡ denotes semantical equivalence. The third equation carries the side-condition that the variable `x` is not free in `g`.

- `(return x) >>= f` ≡ `f x`
- `v >>= return` ≡ `v`
- `(v >>= g) >>= f` ≡ `v >>= (\x -> g x >>= f)`

So far we have seen monads in the abstract; we now turn to some concrete instances. Many ideas can be expressed within the monadic framework, for instance values in `[a]` and `Maybe a`.

```
instance Monad [] where
  l >>= f  = (concat . map f) l
  return x = [x]
```

The monadic nature of lists is made clear by the above instantiation: single values form singleton lists; from the type signature of `>>=`, expression `f` must of type `a -> [b]` and is suitably applied to `l` to yield a value in `[b]`.

```
instance Monad Maybe where
  (Just x) >>= f  = f x
  Nothing  >>= f  = Nothing
  return x        = Just x
```

In order to verify that these are indeed valid monadic instances one must check that the definitions obey the monadic laws.

15.1 I/O

One of the most appreciable practical uses of monads in Haskell programming is for performing I/O. The `IO a` monad is the type of computations returning a value of type `a`; output computations are not expected to return anything, so they are usually typed `IO ()`.

A distinction that must be drawn early is between I/O in terms of strings and that in terms of arbitrary types – modulo their belonging to the classes `Show a` and `Read a` described previously in §13.1. The difference between the two can be seen from the description of the I/O functions in Table 1 and Table 2: the functions in the latter expect the (Haskell) type of the input object to be inferable, or given explicitly. The following examples define functions of type `IO ()` and show the composition of effectful computations using bind.

```
--reads a character line, then prints it
echo  = getLine >>= putStr

--reads an integer, then prints it
echo' = (readLn::IO Integer) >>= (putStr.show)
```

It is useful to have the following standard definition. An example of its use follows.

```
(>>) :: (Monad m) => m a -> m b -> m b
(>>) m n = m >>= \_->n
```

The following programs are equivalent:

- `putStr "1" >>= \x->putStr "2"`
- `putStr "1" >> putStr "2"`

The `do` notation simulates imperative programming style in a purely-functional setting; it is provided as syntactical sugaring. The previous definitions are restated next using this notation. Note that in the new definition of `echo'` a generic output function is used, instead of first casting into a string then printing the string as in the previous definition. The operator `<-` emulates variable assignment.

```
echo  = do x <- getLine
           putStr x

echo' = do x <- (readLn::IO Integer)
           print x
```

15.2 State

When using "pure" functions state must be threaded explicitly throughout the computation. This is unwieldy, so monads together with the `do` notation can be combined to propagate state implicitly.

Imperative programs can be regarded as state transformers. Modelled functionally, they are instances of a function of type `s -> (s,a)`, where `s` is the type of the state and `a` is the output of the function – transformation of the state is a *side-effect* when it does not appear in the function's signature. The following simplistic example shows how to code stateful programs in Haskell. To start one must import the appropriate library and specify what information the state should contain; this is done below in the type `St` that specifies that the state is of type `Bool` and that the return value of functions will be of type `Int`. Then `initVal` specifies a start value; note that its type is made explicit by the accompanying annotation.

```
import Control.Monad.State

type St = State Bool Int
initVal = (return 5)::St
```

The state transformation function is coded next. The definition `stTransf` below flips the Boolean state value, it uses the provided combinators `get` and `put` and returns a dummy integer value. The definition `valTransf` focuses instead on the value returned by the function rather than transforming the state, and invokes `inc` which increments this value. An alternative specification of `inc` is given below it.

```
stTransf :: St -> St
stTransf w = do x <- get
                put $ not x
                return 0

valTransf :: St -> St
valTransf w = do one <- inc w
                 two <- inc w
                 return two

inc :: St -> St
inc w = fmap (+1) w

inc' w = do x <- w
            return $ x+1
```

The reference to the state in `valTransf` may be factored out, and the definition can be restated as follows. Also note that the line `one <- inc w` is redundant, since it `inc` does not influence the state and merely increments the value it received – this is done by the line that follows it too. Note that in general definitions need not be classified into those focusing on state and others on values.

```
valTransf :: St -> St
valTransf = do
               two <- inc
               return two
```

These definitions can be composed into a definition that mixes modification of the state with that of the output value, as shown in `transf`. Note that the output value of `stTransf` has been discarded. The definition is followed by a few test functions to show how the computation is started: the combinator `runState` is applied to a state transformer and an initial state. This combinator returns a pair consisting of the output value and the contents of the state. However, one is usually not interested in the contents of the state: the combinator `evalState` can be used to return only the value of the computation.

```
transf :: St -> St
transf = do stTransf
            x <- valTransf
            return x
```

```
run1 = runState initVal False
run1' = runState (stTransf initVal) False
run2 = runState (transf initVal) False

run = evalState (transf initVal) False
```

16 Practical techniques

Haskell's rules are in place to guard against the introduction of a large class of errors, but in practice it is true both that playing by Haskell's rules may be tedious, and that Haskell's rules are far from sufficient to guarantee correctness. This section describes how the rules can be bent and also complemented.

16.1 Q'n'D

Sometimes programming must be quick and dirty. One can cock a snook at Haskell's type system by using the function `unsafeCoerce`.

```
unsafeCoerce     :: a -> b
unsafePerformIO :: IO a -> a
```

The function `unsafePerformIO`, that prints values from anywhere within a definition, is generally more useful. Note that this collapses the type stratification described earlier, as is evident from its type signature. The following definition sequentially composes together producing output and the evaluation of an expression. Note that this definition takes up the type of the expression that it gets passed; in fact it behaves like the identity function with a side-channel, modulo the first argument.

```
import System.IO.Unsafe

priorPrint :: String -> a -> a
priorPrint str exp =
  seq (unsafePerformIO $ putStr $ str) exp
```

The previous definition used the combinator `seq` which behaves as follows: it evaluates the first argument and returns the second. This is used to offer programmers control over evaluation – they can force strict evaluation, using the strict application operator `$!` defined below.

```
seq :: a -> b -> b

($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

16.2 QuickCheck

QuickCheck [1] is a Haskell library intended to assist in defining Haskell function properties within Haskell, and verifying these properties for a finite collection of values using random testing.

The properties of Haskell functions are themselves encoded as Haskell functions and described using combinators provided by the QuickCheck library. The library also contains definitions that help in describing the distribution of random inputs that will be fed in during testing. It also facilitates the definition of input generators for user-defined types and controlling the size of inputs produced.

An appeal of using QuickCheck is that it incentivates documenting the program using source-level specifications of functions since the specification may be used to test the implementation. Moreover, since the specifications are themselves Haskell functions, both the specification and implementation are expressed in the same formalism; the programmer is not required to learn a new language. A specification is tested by applying the function `quickCheck` to it.

The following code snippets demonstrate building a specification and making it checkable by QuickCheck, that for any `t::Tree a` it is the case that

$$\text{size}\,t > 2(\text{height}\,t)$$

The Haskell encoding of this property is straightforward:

```
prop_SizeHeight :: (Tree Int) -> Bool
prop_SizeHeight t = (size t) > 2 * (height t)
```

Note that despite the property applying to arbitrary types contained within a Tree, the above function has been restricted to a monomorphic definition by means of its type signature. This is necessary in order to indicate to `quickCheck` the appropriate random data generator to invoke. The definition of `height` was given in §12, and `size` is defined below.

```
size :: (Tree a) -> Int
size Leaf = 1
size (Node _ t1 t2)
            = 1 + (size t1) + (size t2)
```

In order to test whether the specified law holds the `quickCheck` function is applied to it to carry out random testing. It is up to the programmer to provide some characterisation for arbitrary values of this type according to some distribution. The following code is adapted from the original paper on QuickCheck [1] and instantiates values of `Tree a` in the class of types for which random values can be synthesised.

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary = sized arbTree
```

The function arbTree generates finite and fully-defined tree values. The values generated are bounded in size by using the combinator sized.

```
arbTree 0 = return Leaf
arbTree n = frequency
  [ (1, return Leaf)
  , (4, liftM3 Node arbitrary
                    (arbTree (n `div` 2))
                    (arbTree (n `div` 2))
    )
  ]
```

The definition of arbTree is parametrised by the maximum size of the tree it is to produce – note that the trees produced are not necessarily balanced, since arbTree's parameter only sets an upper bound. The function liftM3 lifts a pure 3-ary function to be applicable to three monadic values. The distribution of values it produces is controlled by means of the frequency combinator: it is four times more likely to produce an inner node than a leaf. It calls arbitrary to generate random integers for the tree to contain and, in the interest of termination, halves the maximum size expected of subtree values.

Now testing the implementation yields the following:

```
quickCheck prop_SizeHeight
⇝ OK, passed 100 tests.
```

QuickCheck can be tuned in various ways, for instance to carry out more tests or produce more verbose output about the tests made. In the case that a test fails it returns the input values that falsified the property.

One could argue that using QuickCheck increases the scope where bugs may appear since it requires complementing an implementation with a specification and possibly with a definition of arbitrary values of a particular type. As always, care is required on the programmer's part, but because of the richer information being pressed into the source file it is easier to observe errors in both implementation and specification.

Acknowledgements

I thank Stefan Kahrs for providing feedback.

References

[1] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell programs. *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[2] A.D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1994.

[3] S.P. Jones et al. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[4] Simon Thompson. A Logic for Miranda. *Formal Aspects of Computing*, 1:339–365, 1989.

[5] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second edition.

[6] Mads Tofte. Tips for Computer Scientists on Standard ML. Revised version. Obtainable from http://www.itu.dk/people/tofte, April 2008.