A Gentle Introduction to Haskell

Paul Hudak
Yale University
Department of Computer Science

Joseph H. Fasel
University of California
Los Alamos National Laboratory

1 Introduction

This edition of the tutorial has been updated to reflect changes in the Haskell language that will appear in the next version of the Haskell report, 1.3. These changes have been implemented in version 2.2 of the Yale Haskell system and this particular edition of the tutorial has been altered from its original form to reflect these changes. The only significant change in this tutorial is that the section on Input/Output has been updated for the new monadic I/O system.

Our purpose in writing this tutorial is not to teach programming, nor even to teach functional programming. Rather, it is intended to serve as a supplement to the Haskell Report [3], which is otherwise a rather dense technical exposition. Our goal is to provide a gentle introduction to Haskell for someone who has experience with at least one other language, preferably a functional language (even if only an "almost-functional" language such as ML or Scheme). If the reader wishes to learn more about the functional programming style, we highly recommend Bird and Wadler's text *ffntroffufftffon to Funfftffonffl ffroffrffmmffnff* [1], which uses a language sufficiently similar to Haskell to make translation between the two quite easy. For a useful survey of functional programming languages and techniques, including some of the language design principles used in Haskell, see [2].

Our general strategy for introducing language features is this: motivate the idea, define some terms, give some examples, and then point to the Report for details. We suggest, however, that the reader completely ignore the details until this document has been completely read. On the other hand, Haskell's Standard Prelude (in Appendix A of the Report) contains lots of useful examples of Haskell code; we encourage a thorough reading once this tutorial is completed. This will not only give the reader a feel for what real Haskell code looks like, but will also familiarize her with Haskell's standard set of predefined functions and types.

[We have also taken the course of not laying out a plethora of lexical syntax rules at the outset. Rather, we introduce them incrementally as our examples demand, and enclose them in brackets, as with this paragraph. This is in stark contrast to the organization of the Report, although the Report remains the authoritative source for details (references such as "§2.1" refer to sections in the Report).]

Haskell is a *typfffful* programming language[1]: Types are pervasive, and the newcomer is best off becoming well-aware of the full power and complexity of Haskell's type system from the outset. For

[1] A phrase due to Luca Cardelli.

those whose only experience is with relatively "untypeful" languages such as Basic or Lisp, this may be a difficult adjustment; for those familiar with Pascal, Modula, or even ML, the adjustment should be easier but still not insignificant, since Haskell's type system is different and somewhat richer than most. In any case, "typeful programming" is part of the Haskell programming experience, and cannot be avoided.

2 Values, Types, and Other Goodies

Because Haskell is a purely functional language, all computations are done via the evaluation of *ffxprffssffons* (syntactic terms) to yield *vffluffs* (abstract entities that we regard as answers). Every value has an associated *typff*. (Intuitively, we can think of types as sets of values.) Examples of expressions include atomic values such as the integer `5`, the character `'a'`, and the successor function `succ`, as well as structured values such as the list `[1,2,3]` and the pair `('b',4)`.

Just as expressions denote values, *typff ffxprffssffons* are syntactic terms that denote *typff vffluffs* (or just *typffs*). Examples of type expressions include the atomic types `Int` (fixed-precision integers), `Char` (ASCII characters), `Int->Int` (functions mapping `Int` to `Int`), as well as the structured types `[Int]` (homogeneous lists of integers) and `(Char,Int)` (character/integer pairs).

All Haskell values are "first-class"—they may be passed as arguments to functions, returned as results, placed in data structures, etc. Haskell types, on the other hand, are *not* first-class. Types in a sense *ffffsffrffffff* values, and the association of a value with its type is called a *typffnff*. Using the examples of values and types above, we write typings as follows:

```
    5  :: Int
  'a'  :: Char
 succ  :: Int -> Int
[1,2,3] :: [Int]
('b',4) :: (Char,Int)
```

The `::` can be read "has type."

Functions in Haskell are normally defined by a series of *ffqufftffons*. For example, the successor function `succ` can be defined by the single equation:

```
succ n          = n+1
```

An equation is an example of a *fffffflffrfftffon*. Another kind of declaration is a *typff sffffnfftурff ffffffffrfftffon* (§4.4.1), with which we can declare an explicit typing for `succ`:

```
succ            :: Int -> Int
```

We will have much more to say about function definitions in Section 3.

For pedagogical purposes, when we wish to indicate that an expression e_1 evaluates, or "reduces," to another expression or value e_2, we will write:

$$e_1 \quad \Rightarrow \quad e_2$$

For example, note that:

```
succ (succ 3)    ⇒    5
```

Haskell's *stfftffff typff systffm* defines the formal relationship between types and values (§4.1.3). The static type system ensures that Haskell programs are *typff sffffff*; that is, that the programmer has not mismatched types in some way. For example, we cannot generally add together two characters, so the expression `'a'+'b'` is ill-typed. The main advantage of statically typed languages is well-known: All type errors are detected at compile-time. This not only aids the user in reasoning about programs, but also permits a compiler to generate more efficient code (for example, no run-time type tags or tests are required).

The type system also ensures that user-supplied type signatures are correct. In fact, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all,[2] in which case we say that the type system *ffnfffffrs* the correct types for us. Nevertheless, judicious placement of type signatures is a good idea, as we did for `succ`, since it improves readability and helps bring programming errors to light.

[The reader will note that we have capitalized identifiers that denote specific types, such as `Int` and `Char`, but not identifiers that denote values, such as `succ`. This is not just a convention: it is enforced by Haskell's lexical syntax. In fact, the case of the other characters matters, too: `foo`, `fOo`, and `fOO` are all distinct identifiers.]

2.1 Polymorphic Types

Haskell also incorporates *polymorpfffffff* types—types that are universally quantified in some way over *ffll* types. Polymorphic type expressions essentially describe *ffffmfflffffs* of types. For example, (∀a)`[a]` is the family of types consisting of, for every type `a`, the type of lists of `a`. Lists of integers (e.g. `[1,2,3]`), lists of characters (`['a','b','c']`), even lists of lists of integers, etc., are all members of this family. (Note, however, that `[2,'b']` is *not* a valid example, since there is no single type that contains both `2` and `'b'`.)

[Identifiers such as `a` above are called *typff vffrfffffflffs*, and are uncapitalized to distinguish them from specific types such as `Int`. Furthermore, since Haskell has only universally quantified types, there is no need to explicitly write out the symbol for universal quantification, and thus we simply write `[a]` in the example above. In other words, all type variables are implicitly universally quantified.]

Lists are a commonly used data structure in functional languages, and are a good vehicle for explaining the principles of polymorphism. The list `[1,2,3]` in Haskell is actually shorthand for the list `1:(2:(3:[]))`, where `[]` is the empty list and `:` is the infix operator that adds its first argument to the front of its second argument (a list).[3] Since `:` is right associative, we can also write this list as `1:2:3:[]`.

As an example of a user-defined function that operates on lists, consider the problem of counting the number of elements in a list:

[2] With a few exceptions to be described later.

[3] `:` and `[]` are like Lisp's `cons` and `nil`, respectively.

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

This definition is almost self-explanatory. We can read the equations as saying: "The length of the empty list is 0, and the length of a list whose first element is `x` and remainder is `xs` is 1 plus the length of `xs`." (Note the naming convention used here; `xs` is the plural of `x`, and should be read that way.)

Although intuitive, this example highlights an important aspect of Haskell that is yet to be explained: *pfftffrn mfftfffffnff*. The left-hand sides of the equations contain *pfftffrns* such as `[]` and `x:xs`. In a function application these patterns are *mfftffffff* against actual parameters in a fairly intuitive way (`[]` only matches the empty list, and `x:xs` will successfully match any list with at least one element, binding `x` to the first element and `xs` to the rest of the list). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results.

Defining functions by pattern matching is quite common in Haskell, and the user should become familiar with the various kinds of patterns that are allowed; we will return to this issue in Section 3.14.

`length` is also an example of a *polymorpfffff ffunfftffon*. It can be applied to a list containing elements of any type. For example:

```
length [1,2,3]          ⇒    3
length ['a','b','c']    ⇒    3
length [[],[],[]]       ⇒    3
```

Here are two other useful polymorphic functions on lists that will be used later:

```
head                    :: [a] -> a
head (x:xs)             =  x

tail                    :: [a] -> [a]
tail (x:xs)             =  xs
```

With polymorphic types, we find that some types are in a sense strictly *morff ffffnffrffl* than others. For example, the type `[a]` is more general than `[Char]`. In other words, the latter type can be derived from the former by a suitable substitution for `a`. With regard to this generalization ordering, Haskell's type system possesses two important properties: First, every well-typed expression is guaranteed to have a unique *prffnffffpffl* type (explained below), and second, the principal type can be *ffnffffrrffff* automatically (§4.1.3). In comparison to a *monomorpfffffffflly typffff* language such as Pascal, the reader will find that *polymorpffffsm* improves expressiveness, and *typff ffnffffrffnffff* lessens the burden of types on the programmer.

An expression's or function's principal type is the least general type that, intuitively, "contains all instances of the expression." For example, the principal type of `head` is `[a]->a`; the types

`[b]->a`, `a->a`, or even `a` are *too ffffnffrffl*, whereas something like `[Int]->Int` is *too spfffffffiff*. The existence of unique principal types is the hallmark feature of the *fffnfflffy-fffflnffr typff systffm*, which forms the basis of the type systems of Haskell, ML, Miranda,[4] and several other (mostly functional) languages.

2.2 User-Defined Types

We can define our own types in Haskell using a `data` declaration, which we introduce via a series of examples (§4.2.1).

An important predefined type in Haskell is that of truth values:

```
data Bool               = False | True
```

The type being defined here is `Bool`, and it has exactly two values: `True` and `False`. `Bool` is an example of a (nullary) *typff ffonstrufftor*, and `True` and `False` are (also nullary) *fffftff ffonstrufftors* (or just *ffonstrufftors*, for short).

Similarly, we might wish to define a color type:

```
data Color              = Red | Green | Blue | Indigo | Violet
```

Both `Bool` and `Color` are examples of *ffnumffrfftffff typffs*, since they consist of a finite number of nullary data constructors.

Here is an example of a type with just one data constructor:

```
data Point a            = Pt a a
```

Because of the single constructor, a type like `Point` is often called a *tuplff typff*, since it is essentially just a cartesian product (in this case binary) of other types.[5] In contrast, multi-constructor types, such as `Bool` and `Color`, are called (disjoint) *unffon* types.

More importantly, however, `Point` is an example of a *polymorpffffff* type: for any type t, it defines the type of cartesian points that use t as the coordinate type. `Point` can now be seen clearly as a unary type constructor, since from the type t it constructs a new type `Point` t. (In the same sense, using the list example given earlier, `[_]` is also a type constructor (where we have used "`_`" to denote the missing argument): given any type t we can "apply" `[_]` to yield a new type `[`t`]`. Similarly, `_->_` is a type constructor: given two types t and u, t`->`u is the type of functions mapping elements of type t to elements of type u.)

Note that the type of the binary constructor `Pt` is `a -> a -> Point a`, and thus the following typings are valid:

[4] "Miranda" is a trademark of Research Software, Ltd.

[5] Tuples are somewhat like *records* in other languages, except that the elements are positional, rather than having names (labels) associated with them.

```
Pt  2.0  3.0          :: Point Float
Pt  'a'  'b'          :: Point Char
Pt True False         :: Point Bool
```

On the other hand, an expression such as `Pt 'a' 1` is ill-typed.

It is important to distinguish between applying a *ffonstrufftor* to yield a *vffluff*, and applying a *typff ffonstrufftor* to yield a *typff*; the former happens at run-time and is how we compute things in Haskell, whereas the latter happens at compile-time and is part of the type system's process of ensuring type safety.

2.3 Recursive Types

Types can also be recursive, as in:

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

Here we have defined a polymorphic binary tree type whose elements are either leaf nodes containing a value of type `a`, or internal nodes ("branches") containing (recursively) two sub-trees.

When reading data declarations such as this, remember that `Tree` is a type constructor, whereas `Branch` and `Leaf` are data constructors. Aside from establishing a connection between these constructors, the above declaration is essentially defining the following types for `Branch` and `Leaf`:

```
Branch                  :: Tree a -> Tree a -> Tree a
Leaf                    :: a -> Tree a
```

With this example we have defined a type sufficiently rich to allow defining some interesting (recursive) functions that use it. For example, suppose we wish to define a function `fringe` that returns a list of all the elements in the leaves of a tree from left to right. It's usually helpful to write down the type of new functions first; in this case we see that the type should be `Tree a -> [a]`. That is, `fringe` is a polymorphic function that, for any type `a`, maps trees of `a` into lists of `a`. A suitable definition follows:

```
fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Where `++` is the infix operator that concatenates two lists (its full definition will be given in Section 3.2). As with the `length` example given earlier, `fringe` is defined using pattern matching, except that here we see patterns involving user-defined constructors: `Leaf` and `Branch`. [Note that the formal parameters are easily identified as the ones beginning with lower-case letters.]

2.4 Type Synonyms

For convenience, Haskell provides a way to define *typff synonyms*; i.e. names for commonly used types. Type synonyms are created using a `type` declaration (§4.2.2). Here are several examples:

```
type String             = [Char]
type Person             = (Name,Address)
type Name               = String
data Address            = None | Addr String
```

Type synonyms do not define new types, but simply give new names for existing types. For example, the type `Person -> Name` is precisely equivalent to `(String,Address) -> String`. The new names are often shorter than the types they are synonymous with, but this is not the only purpose of type synonyms: they can also improve readability of programs by being more mnemonic; indeed, the above examples highlight this. We can even give new names to polymorphic types:

```
type AssocList a b              = [(a,b)]
```

This is the type of "association lists" which associate values of type `a` with those of type `b`.

2.5 Built-in Types Are Not Special

Earlier we introduced several "built-in" types such as lists, tuples, integers, and characters. We have also shown how new user-defined types can be defined. Aside from special syntax, are the built-in types in any way more special than the user-defined ones? The answer is *no*. The special syntax is for convenience and for consistency with historical convention, but has no semantic consequence.

We can emphasize this point by considering what the type declarations would look like for these built-in types if in fact we were allowed to use the special syntax in defining them. For example, the `Char` type might be written as:

```
data Char       = 'a' | 'b' | 'c' | ...         -- This is not valid
                | 'A' | 'B' | 'C' | ...         -- Haskell code!
                | '1' | '2' | '3' | ...
                ...
```

These constructor names are not syntactically valid; to fix them we would have to write something like:

```
data Char       = Ca | Cb | Cc | ...
                | CA | CB | CC | ...!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

`the typefont used in this manuscript.`

[8]Fixity declarations must only appear at the very beginning of a Haskell *module*, as will be described in Section 6.

Both of these specify right-associativity, the first with a precedence level of 5, the other 9. Left associativity is specified via `infixl`, and non-associativity by `infix`. Also, the fixity of more than one operator may be specified with the same fixity declaration. If no fixity declaration is given for a particular operator, it defaults to `infixl 9`. (See §5.7 for a detailed definition of the associativity rules.)

3.3 Functions are Non-strict

Suppose `bot` is defined by:

```
bot                     = bot
```

In other words, `bot` is a non-terminating expression. Abstractly, we denote the *vffluff* of a non-terminating expression as $\perp$ (read "bottom"). Expressions that result in some kind of a run-time error, such as `1/0`, also have this value.

A function `f` is said to be *strfffft* if, when applied to a nonterminating expression, it also fails to terminate. In other words, `f` is strict iff the value of `f bot` is $\perp$. For most programming languages, *ffll* functions are strict. But this is not so in Haskell. As a simple example, consider `const1`, the constant 1 function, defined by:

```
const1 x                = 1
```

The value of `const1 bot` in Haskell is `1`. Operationally speaking, since `const1` does not "need" the value of its argument, it never attempts to evaluate it, and thus never gets caught in a nonterminating computation. For this reason, non-strict functions are also called "lazy functions," and are said to evaluate their arguments "lazily," or "by need."

Since error and nonterminating values are semantically the same in Haskell, the above argument also holds for errors. For example, `const1 (1/0)` also evaluates properly to `1`.

Non-strict functions are extremely useful in a variety of contexts. The main advantage is that they free the programmer from many concerns about evaluation order. Computationally expensive values may be passed as arguments to functions without fear of them being computed if they are not needed. An important example of this is a possibly *ffnfinfftff* data structure.

3.4 "Infinite" Data Structures

One advantage of the non-strict nature of Haskell is that data constructors are non-strict, too. This should not be surprising, since constructors are really just a special kind of function (the distinguishing feature being that they can be used in pattern matching). For example, the constructor for lists, `(:)`, is non-strict.

Non-strict constructors permit the definition of (conceptually) *ffnfinfftff* data structures. Here is an infinite list of ones:

```
ones                    = 1 : ones
```

Perhaps more interesting is the function `numsFrom`:

```
numsFrom n              = n : numsFrom (n+1)
```

Thus `numsFrom n` is the infinite list of successive integers beginning with `n`. From it we can construct an infinite list of squares:

```
squares                 = map (^2) (numsfrom 0)
```

(Note the use of a section; `^` is the infix exponentiation operator.)

Of course, eventually we expect to extract some finite portion of the list for actual computation, and there are lots of predefined functions in Haskell that do this sort of thing: `take`, `takeWhile`, `filter`, and others (see the portion of the Standard Prelude called `PreludeList`). For example, `take` removes the first `n` elements from a list:

```
take 5 squares    ⇒    [0,1,4,9,16]
```

The definition of `ones` above is an example of a *ffffrffulffr lffst*. In most circumstances this has an important impact on efficiency, since an implementation can be expected to implement the list as a true circular structure, thus saving space.

For another example of the use of circularity, the Fibonacci sequence can be computed efficiently as the following infinite sequence:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

where `zip` is a Standard Prelude function that returns the pairwise interleaving of its two list arguments:

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

Note how `fib`, an infinite list, is defined in terms of itself, as if it were "chasing its tail." Indeed, we can draw a picture of this computation as shown in Figure 1a.

For another application of infinite lists, see Section 4.4.

3.5 The Error Function

Haskell has a built-in function called `error` whose type is `String->a`. This is a somewhat odd function: From its type it looks as if it is returning a value of a polymorphic type about which it knows nothing, since it never receives a value of that type as an argument!

In fact, there *ffs* one value "shared" by all types: $\perp$. Indeed, semantically that is exactly what value is always returned by `error` (recall that all errors have value $\perp$). However, we can expect that a reasonable implementation will print the string argument to `error` for diagnostic purposes. Thus this function is useful when we wish to terminate a program when something has "gone wrong."