Why Haskell matters

From HaskellWiki

Contents

1 What are functional programming languages?

Programming languages such as C/C++/Java/Python are called imperative programming languages because they consist of sequences of actions. The programmer quite explicitly tells the computer how to perform a task, step-by-step. Functional programming languages work differently. Rather than performing actions in a sequence, they evaluate expressions.

1.1 The level of abstraction

There are two areas that are fundamental to programming a computer - resource management and sequencing. Resource management (allocating registers and memory) has been the target of vast abstraction, most new languages (imperative as well as functional) have implemented garbage collection to remove resource management from the problem, and lets the programmer focus on the algorithm instead of the book-keeping task of allocating memory. Sequencing has also undergone some abstraction, although not nearly to the same extent. Imperative languages have done so by introducing new keywords and standard libraries. For example, most imperative languages have special syntax for constructing several slightly different loops, you no longer have to do all the tasks of managing these loops yourself. But imperative languages are based upon the notion of sequencing - they can never escape it completely. The only way to raise the level of abstraction in the sequencing area for an imperative language is to introduce more keywords or standard functions, thus cluttering up the language. This close relationship between imperative languages and the task of sequencing commands for the processor to execute means that imperative languages can never rise above the task of sequencing, and as such can never reach the same level of abstraction that functional programming languages can.

In Haskell, the sequencing task is removed. You only care what the program is to compute not how or when it is computed. This makes Haskell a more flexible and easy to use language. Haskell tends to be part of the solution for a problem, not a part of the problem itself.

1.2 Functions and side-effects in functional languages

Functions play an important role in functional programming languages. Functions are considered to be values just like integers or strings. A function can return another function, it can take a function as a parameter, and it can even be constructed by composing functions. This offers a stronger "glue" to combine the modules of your program. A function that evaluates some expression can take part of the computation as an argument for instance, thus making the function more modular. You could also have a function construct another function. For instance, you could define a function "differentiate" that will differentiate a given function numerically. So if you then have a function "f" you could define "f' = differentiate f", and use it like you would normally in a mathematical context. These types of functions are called *higher order functions*.

Here is a short Haskell example of a function numOf that counts the number of elements in a list that satisfy a certain property.

```
numOf p xs = length (filter p xs)
```

We will discuss Haskell syntax later, but what this line says is just "To get the result, filter the list xs by the test p and compute the length of the result". Now p is a function that takes an element and returns True or False determining whether the element passes or fails the test. So numOf is a higher order function, some of the functionality is passed to it as an argument. Notice that filter is also a higher order function, it takes the "test function" as an argument. Let's play with this function and define some more specialized functions from it.

```
numOfEven xs = numOf even xs
```

Here we define the function numOfEven which counts the number of even elements in a list. Note that we do not need to explicitly declare xs as a parameter. We could just as well write numOfEven = numOf even. A very clear definition indeed. But we'll explicitly type out the parameters for now.

Let's define a function which counts the number of elements that are greater or equal to 5 :

```
numOfGE5 xs = numOf (>=5) xs
```

Here the test function is just ">=5" which is passed to numOf to give us the functionality we need.

Hopefully you should now see that the modularity of functional programming allows us to define a generic functions where some of the functionality is passed as an argument, which we can later use to define shorthands for any specialized functions. This small example is somewhat trivial, it wouldn't be too hard to re-write the function definition for all the functions above, but for more complex functions this comes in handy. You can, for instance, write only one function for traversing an auto-balancing binary tree and have it take some of the functionality as a parameter (for instance the comparison function). This would allow you to traverse the tree for any data type by simply providing the relevant comparison function for your needs. Thus you can expend some effort in making sure the general function is correct, and then all the specialized functions will also be correct. Not to mention you wouldn't have to copy and paste code all over your project. This concept is possible in some imperative languages as well. In some object oriented languages you often have to provide a "Comparator object" for trees and other standard data structures. The difference is that the Haskell way is a lot more intuitive and elegant (creating a separate type just for comparing two other types and then passing an object of this type is hardly an elegant way of doing it), so it's more likely to be used frequently (and not just in the standard libraries).

A central concept in functional languages is that the result of a function is determined by its input, and only by its input. There are no side-effects! This extends to variables as well - *variables in Haskell do not vary*. This may sound strange if you're used to imperative programming (where *most* of the code consists of changing the "contents" of a variable), but it's really quite natural. A variable in Haskell is a name that is bound to some value, rather than an abstraction of some low-level concept of a memory cell like in imperative languages. When variables are thought of as short-hands for values (just like they are in mathematics), it makes perfect sense that variable updates are not allowed. You wouldn't expect "4 = 5" to be a valid assignment in *any* language, so it's really quite strange that "x = 4; x = 5" is. This is often hard to grasp for programmers who are very used to imperative languages, but it isn't as strange as it first seems. So when you start thinking things like "This is too weird, I'm going back to C++!", try to force yourself to continue learning Haskell - you'll be glad you did.

Removing side-effects from the equation allows expressions to be evaluated in any order. A function will always return the same result if passed the same input - no exceptions. This determinism removes a whole class of bugs found in imperative programs. In fact, you could even argue that *most* bugs in large systems can be traced back to side-effects - if not directly caused by them, then caused by a flawed design that relies on side-effects. This means that functional programs tend to have far fewer bugs than imperative ones.

1.3 Conclusion

Because functional languages are more intuitive and offer more and easier ways to get the job done, functional programs tend to be shorter (usually between 2 to 10 times shorter). The semantics are most often a lot closer to the problem than an imperative version, which makes it easier to verify that a function is correct. Furthermore Haskell doesn't allow side-effects, which leads to fewer bugs. Thus Haskell programs are easier to write, more robust, and easier to maintain.

2 What can Haskell offer the programmer?

Haskell is a modern general purpose language developed to incorporate the collective wisdom of the functional programming community into one elegant, powerful and general language.

2.1 Purity

Unlike some other functional programming languages Haskell is pure. It doesn't allow *any* side-effects. This is probably the most important feature of Haskell. We've already briefly discussed the benefits of pure, side-effect free, programming - and there's not much more we can say about that. You'll need to experience it yourself.

2.2 Laziness

Another feature of Haskell is that it is lazy (technically speaking, it's "non-strict"). This means that nothing is evaluated until it has to be evaluated. You could, for instance, define an infinite list of primes without ending up in infinite recursion. Only the elements of this list that are actually used will be computed. This allows for some very elegant solutions to many problems. A typical pattern of solving a problem would be to define a list of all possible solutions and then filtering away the illegal ones. The remaining list will then only contain legal solutions. Lazy evaluation makes this operation very clean. If you only need one solution you can simply extract the first element of the resulting list - lazy evaluation will make sure that nothing is needlessly computed.

2.3 Strong typing

Furthermore Haskell is strongly typed, this means just what it sounds like. It's impossible to inadvertently convert a Double to an Int, or follow a null pointer. This also leads to fewer bugs. It might be a pain in the neck in the rare cases where you need to convert an Int to a Double explicitly before performing some operation, but in practice this doesn't happen often enough to become a nuisance. In fact, forcing each conversion to be explicit often helps to highlight problem code. In other languages where these conversions are invisible, problems often arise when the compiler treats a double like an integer or, even worse, an integer like a pointer.

Unlike other strongly typed languages types in Haskell are automatically inferred. This means that you very rarely have to declare the types of your functions, except as a means of code documentation. Haskell will look at how you use the variables and figure out from there what type the variable should be - then it will all be type-checked to ensure there are no type-mismatches. Python has the notion of "duck typing", meaning "If it walks and talks like a duck, it's a duck!". You could argue that Haskell has a much better form of duck typing. If a value walks and talks like a duck, then it will be considered a duck through type inference, but unlike Python the compiler will also catch errors if later on it tries to bray like a donkey! So you get the benefits of strong typing (bugs are caught at compile-time, rather than run-time) without the hassle that comes with it in other languages. Furthermore Haskell will always infer the most general type on a variable. So if you write, say, a sorting function without a type declaration, Haskell will make sure the function will work for all values that can be sorted.

Compare how you would do this in certain object oriented languages. To gain polymorphism you would have to use some base class, and then declare your variables as instances of subclasses to this base class. It all amounts to tons of extra work and ridiculously complex declarations just to proclaim the existence of a variable. Furthermore you would have to perform tons of type conversions via explicit casts - definitely not a particularly elegant solution. If you want to write a polymorphic function in these object oriented languages you would probably declare the parameters as an object of a global base class (like "Object" in Java), which essentially allows the programmer to send anything into the function, even objects which can't logically be passed to the function. The end result is that most functions you write in these languages are not general, they only work on a single data type. You're also moving the error checking from compile-time to run-time. In large systems where some of the functionality is rarely used, these bugs might never be caught until they cause a fatal crash at the worst possible time.

Haskell provides an elegant, concise and safe way to write your programs. Programs will not crash unexpectedly, nor produce strangely garbled output.

2.4 Elegance

Another property of Haskell that is very important to the programmer, even though it doesn't mean as much in terms of stability or performance, is the elegance of Haskell. To put it simply: stuff just works like you'd expect it to.

To highlight the elegance of Haskell we shall now take a look at a small example. We choose QuickSort-inspired filtering sort because it's a simple algorithm that is actually useful. We will look at two versions - one written in C++, an imperative language, and one written in Haskell. Both versions use only the functionality available to the programmer without importing any extra modules (otherwise we could just call "sort" in each language's standard library and be done with it!). Thus, we use the standard sequence primitives of each language (a "list" in Haskell and an "array" in C++). Both versions must also be polymorphic (which is done "automatically" in Haskell, and with templates in C++). Both versions must use the same recursive algorithm.

Please note that this is *not* intended as a definite comparison between the two languages. It's intended to show the elegance of Haskell, the C++ version is only included for comparison (and would be coded quite differently if you used the standard QuickSort algorithm or the Standard Template Libraries (STL), for example).

```
template <typename T>
void qsort (T *result, T *list, int n)
{
    if (n == 0) return;
    T *smallerList, *largerList;
    smallerList = new T[n];
    largerList = new T[n];
    T pivot = list[0];
    int numSmaller=0, numLarger=0;
    for (int i = 1; i < n; i++)
        if (list[i] < pivot)
            smallerList[numSmaller++] = list[i];
        else
            largerList[numLarger++] = list[i];

    qsort(smallerList,smallerList,numSmaller);
    qsort(largerList,largerList,numLarger);

    int pos = 0;
    for ( int i = 0; i < numSmaller; i++)
        result[pos++] = smallerList[i];

    result[pos++] = pivot;

    for ( int i = 0; i < numLarger; i++)
        result[pos++] = largerList[i];

    delete [] smallerList;
    delete [] largerList;
};
```

We will not explain this code further, just note how complex and difficult it is to understand at a glance, largely due to the programmer having to deal with low-level details which have nothing to do with the task at hand. Now, let's take a look at a Haskell version of FilterSort, which might look a something like this.

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
    where less = filter (<x)  xs
          more = filter (>=x) xs
```

(This implementation has very poor runtime and space complexity, but that can be improved, at the expense of some of the elegance (http://augustss.blogspot.com/2007/08/quicksort-in-haskell-quicksort-is.html) .)

Let's dissect this code in detail, since it uses quite a lot of Haskell syntax that you might not be familiar with.

The first line is a type signature. It declares "qsort" to be function that takes a list "[a]" as input and returns ("->") another list "[a]". "a" is a type variable (vaguely similar to a C++ template declaration), and "(Ord a)" is a constraint that means that only types that have an ordering are allowed. This function is a generic ("template") function, that can sort any list of pairwise-comparable objects. The phrase "(Ord a) => [a] -> [a]" means "if the type 'a' is ordered, than a list of 'a' can be passed in, and another list of 'a' will come out."

The function is called qsort and takes a list as a parameter. We define a function in Haskell like so: funcname a b c = expr, where funcname is the name of the function, a, b, and, c are the parameters and expr is the expression to be evaluated (most often using the parameters). Functions are called by simply putting the function name first and then the parameter(s). Haskell doesn't use parenthesis for function application. Functions simply bind more tightly than anything else, so "f 5 * 2", for instance, would apply f to 5 and then multiply by 2, if we wanted the multiplication to occur before the function application then we would use parenthesis like so "f (5*2)".

Let's get back to FilterSort. First we see that we have two definitions of the functions. This is called pattern matching and we can briefly say that it will test the argument passed to the function top-to-bottom and use the first one that matches. The first definition matches against [] which in Haskell is the empty list (a list of 1,2 and 3 is [1,2,3] so it makes sense that an empty list is just two brackets). So when we try to sort an empty list, the result will be an empty list. Sounds reasonable enough, doesn't it? The second definition pattern matches against a list with at least one element. It does this by using (x:xs) for its argument. The "cons" operator is (:) and it simply puts an element in front of a list, so that 0 : [1,2,3] returns [0,1,2,3]. Pattern matching against (x:xs) is a match against the list with the head x and the tail xs (which may or may not be the empty list). In other words, (x:xs) is a list of at least one element. So since we will need to use the head of the list later, we can actually extract this very elegantly by using pattern matching. You can think of it as naming the contents of the list. This can be done on any data construct, not just a list. It is possible to pattern match against an arbitrary variable name and then use the head function on that to retrieve the head of the list. Now if we have a non empty list, the sorted list is produced by sorting all elements that are smaller than x and putting that in front of x, then we sort all elements larger than x and put those at the end. We do this by using the list concatenation operator ++. Notice that x is not a list so the ++ operator won't work on it alone, which is why we make it a singleton-list by putting it inside brackets. So the function reads "To sort the list, sandwich the head between the sorted list of all elements smaller than the head, and the sorted list of all elements larger than the head". Which could very well be the original algorithm description. This is very common in Haskell. A function definition usually resembles the informal description of the function very closely. This is why I say that Haskell has a smaller semantic gap than other languages.

But wait, we're not done yet! How is the list less and more computed? Well, remember that we don't care about sequencing in Haskell, so we've defined them below the function using the where notation (which allows any definitions to use the parameters of the function to which they belong). We use the standard prelude function filter, I won't elaborate too much on this now, but the line less = filter (<x) xs will use filter (<x) xs to filter the list xs. You can see that we actually pass the function which will be used to filter the list to filter, an example of higher order functions. The function (<x) should be read out "the function 'less than x'" and will return True if an element passed to it is less than x (notice how easy it was to construct a function on the fly, we put the expression "<x", "less than x", in parenthesis and sent it off to the function - functions really are just another value!). All elements that pass the test are output from the filter function and put inside less. In a same way (>=x) is used to filter the list for all elements larger than or equal to x.

Now that you've had the syntax explained to you, read the function definition again. Notice how little time it takes to get an understanding about what the function does. The function definitions in Haskell explain what it computes, not how.

If you've already forgotten the syntax outlined above, don't worry! We'll go through it more thoroughly and at a slower pace in the tutorials. The important thing to get from this code example is that Haskell code is elegant and intuitive.

2.5 Haskell and bugs

We have several times stated that various features of Haskell help fight the occurrence of bugs. Let's recap these.

Haskell programs have fewer bugs because Haskell is:

- **Pure**. There are no side effects.
- **Strongly typed**. There can be no dubious use of types. And No Core Dumps!
- **Concise**. Programs are shorter which make it easier to look at a function and "take it all in" at once, convincing yourself that it's correct.
- **High level**. Haskell programs most often reads out almost exactly like the algorithm description. Which makes it easier to verify that the function does what the algorithm states. By coding at a higher level of abstraction, leaving the details to the compiler, there is less room for bugs to sneak in.
- **Memory managed**. There's no worrying about dangling pointers, the Garbage Collector takes care of all that. The programmer can worry about implementing the algorithm, not book-keeping of the memory.
- **Modular**. Haskell offers stronger and more "glue" to compose your program from already developed modules. Thus Haskell programs can be more modular. Often used modular functions can thus be proven correct by induction. Combining two functions that are proven to be correct, will also give the correct result (assuming the combination is correct).

Furthermore most people agree that you just think differently when solving problems in a functional language. You subdivide the problem into smaller and smaller functions and then you write these small (and "almost-guaranteed-to-be-correct") functions, which are composed in various ways to the final result. There just isn't any room for bugs!

3 Haskell vs OOP

The great benefit of Object Oriented Programming is rarely that you group your data with the functions that act upon it together into an object - it's that it allows for great data encapsulation (separating the interface from implementation) and polymorphism (letting a whole set of data types behave the same way). However:

Data encapsulation and polymorphism are not exclusive to OOP!

Haskell has tools for abstracting data. We can't really get into it without first going through the module system and how abstract data types (ADT) work in Haskell, something which is well beyond the scope of this essay. We will therefore settle for a short description of how ADTs and polymorphism works in Haskell.

Data encapsulation is done in Haskell by declaring each data type in a separate module, from this module you only export the interface. Internally there might be a host of functions that touch the actual data, but the interface is all that's visible from outside of the module. Note that the data type and the functions that act upon the data type are not grouped into an "object", but they are (typically) grouped into the same module, so you can choose to only export certain functions (and not the constructors for the data type) thus making these functions the only way to manipulate the data type - "hiding" the implementation from the interface.

Polymorphism is done by using something called type classes. Now, if you come from a C++ or Java background you might associate classes with something resembling a template for how to construct an object, but that's not what they mean in Haskell. A type class in Haskell is really just what it sounds like. It's a set of rules for determining whether a type is an instance of that class. So Haskell separates the class instantiation and the construction of the data type. You might declare a type "Porsche", to be an instance of the "Car" type class, say. All functions that can be applied onto any other member of the Car type class can then be applied to a Porsche as well. A class that's included with Haskell is the Show type class, for which a type can be instantiated by providing a show function, which converts the data type to a String. Consequently almost all types in Haskell can be printed onto the screen by applying show on them to convert them to a String, and then using the relevant IO action (more on IO in the tutorials). Note how similar this is to the the object notion in OOP when it comes to the polymorphism aspect. The Haskell system is a more intuitive system for handling polymorphism. You won't have to worry about inheriting in the correct hierarchical order or to make sure that the inheritance is even sensible. A class is just a class, and types that are instances of this class really doesn't have to share some parent-child inheritance relationship. If your data type fulfills the requirements of a class, then it can be instantiated in that class. Simple, isn't it? Remember the QuickSort example? Remember that I said it was polymorphic? The secret behind the polymorphism in qsort is that it is defined to work on any type in the Ord type class (for "Ordered"). Ord has a set of functions defined, among them is "<" and ">=" which are sufficient for our needs because we only need to know whether an element is smaller than x or not. So if we were to define the Ord functions for our Porsche type (it's sufficient to implement, say, <= and ==, Haskell will figure out the rest from those) in an instantiation of the Ord type class, we could then use qsort to sort lists of Porsches (even though sorting Porsches might not make sense). Note that we never say anything about which classes the elements of the list must be defined for, Haskell will infer this automatically from just looking at which functions we have used (in the qsort example, only "<" and ">=" are relevant).

So to summarize: Haskell does include mechanisms for data encapsulation that match or surpass those of OOP languages. The only thing Haskell does not provide is a way to group functions and data together into a single "object" (aside from creating a data type which includes a function - remember, functions are data!). This is, however, a very minor problem. To apply a function to an object you would write "func obj a b c" instead of something like "obj.func a b c".

4 Modularity

A central concept in computing is modularity. A popular analogy is this: say you wanted to construct a wooden chair. If you construct the parts of it separately, and then glue them together, the task is solved easily. But if you were to carve the whole thing out of a solid piece of wood, it would prove to be quite a bit harder. John Hughes had this to say on the topic in his paper Why Functional Programming Matters (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html)

"Languages which aim to improve productivity must support modular programming well. But new scope rules and mechanisms for separate compilation are not enough - modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming, a language must provide good glue.

Functional programming languages provide two new kinds of glue - higher-order functions and lazy evaluation."

5 The speed of Haskell

Let me first state clearly that the following only applies to the general case in which speed isn't absolutely critical, where you can accept a few percent longer execution time for the benefit of reducing development time greatly. There are cases when speed is the primary concern, and then the following section will not apply to the same extent.

Now, some C++ programmers might claim that the C++ version of QuickSort above is probably a bit faster than the Haskell version. And this might be true. For most applications, though, the difference in speed is so small that it's utterly irrelevant. For instance, take a look at the Computer Language Benchmarks Game (http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=all) , where Haskell compares favorably to most of the so called "fast" languages. Now, these benchmarks don't prove all that much about real-world performance, but they do show that Haskell isn't as slow as some people think. At the time of writing it's in 4th position, only slightly behind C and C++.

Almost all programs in use today have a fairly even spread of processing time among its functions. The most notable exceptions are applications like MPEG encoders, and artificial benchmarks, which spend a large portion of their execution time within a small portion of the code. If you *really* need speed at all costs, consider using C instead of Haskell.

There's an old rule in computer programming called the "80/20 rule". It states that 80% of the time is spent in 20% of the code. The consequence of this is that any given function in your system will likely be of minimal importance when it comes to optimizations for speed. There may be only a handful of functions important enough to optimize. These important functions could be written in C (using the excellent foreign function interface in Haskell). The role of C could, and probably will, take over the role of assembler programming - you use it for the really time-critical bits of your system, but not for the whole system itself.

We should continue to move to higher levels of abstraction, just like we've done before. We should trade application speed for increased productivity, stability and maintainability. Programmer time is almost always more expensive than CPU time. We aren't writing applications in assembler anymore for the same reason we shouldn't be writing applications in C anymore.

Finally remember that algorithmic optimization can give much better results than code optimization. For theoretical examples when factors such as development times and stability doesn't matter, then sure, C is often faster than Haskell. But in the real world development times *do* matter, this isn't the case. If you can develop your Haskell application in one tenth the time it would take to develop it in C (from experience, this is not at all uncommon) you will have lots of time to profile and implement new algorithims. So in the "real world" where we don't have infinite amounts of time to program our applications, Haskell programs can often be much faster than C programs.

6 Epilogue

So if Haskell is so great, how come it isn't "mainstream"? Well, one reason is that the operating system is probably written in C or some other imperative language, so if your application mainly interacts with the internals of the OS, you may have an easier time using imperative languages. Another reason for the lack of Haskell, and other functional languages, in mainstream use is that programming languages are rarely thought of as tools (even though they are). To most people their favorite programming language is much more like religion - it just seems unlikely that any other language exists that can get the job done better and faster.

There is an essay by Paul Graham called Beating the Averages (http://www.paulgraham.com/avg.html) describing his experience using Common Lisp, another functional language, for an upstart company. In it he uses an analogy which he calls "The Blub Paradox".

It goes a little something like this: If a programmer's favorite language is Blub, which is positioned somewhere in the middle of the "power spectrum", he can most often only identify languages that are lower down in the spectrum. He can look at COBOL and say "How can anyone get anything done in that language, it doesn't have feature x", x being a feature in Blub.

However, this Blub programmer has a harder time looking the other way in the spectrum. If he examines languages that are higher up in the power spectrum, they will just seem "weird" because the Blub programmer is "thinking in Blub" and can not possibly see the uses for various features of more powerful languages. It goes without saying that this inductively leads to the conclusion that to be able to compare all languages you'll need to position yourself at the top of the power spectrum. It is my belief that functional languages, almost by definition, are closer to the top of the power spectrum than imperative ones. So languages can actually limit a programmers frame of thought. If all you've ever programmed is Blub, you may not see the limitations of Blub - you may only do that by switching to another level which is more powerful.

One of the reasons the mainstream doesn't use Haskell is because people feel that "their" language does "everything they need". And of course it does, because they are thinking in Blub! Languages aren't just technology, it's a way of thinking. And if you're not thinking in Haskell, it is very hard to see the use of Haskell - even if Haskell would allow you to write better applications in a shorter amount of time!

Hopefully this article has helped you break out of the Blub paradox. Even though you may not yet "think in Haskell", it is my hope that you are at least aware of any limitations in your frame of thought imposed by your current "favorite" language, and that you now have more motivation to expand it by learning something new.

If you are committed to learn a functional language, to get a better view of the power spectrum, then Haskell is an excellent candidate.