1 Haskell or Why Haskell ?

Haskell / Why Haskell ?

1. Haskell as a functional programming language

   Haskell is an advanced purely-functional programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language [**?**]. It is one of the popular functional programming languages [**?**]. Haskell is widely used in the industry [**?**].

2. Haskell as a tool for embedding domain specific languages

   (a) Monads
   Each language has a flow of control, a model how programs are executed. Many languages for example Java have a top-down sequential execution approach. But very few languages allow control flow modelling, for example in a functional language how the propagation of side effects is handled. A monad provides the ability to customize and develop your own model from one step to another, how side effects are handled. [**?**]

   Why Haskell is Good for Embedded Domain Specific Languages Domain Specific Languages (DSLs) are attracting some attention these days. They have always been around, of course: Emacs Lisp is a DSL, as are the various dialects of Visual Basic embedded in MS Office applications. And of course Unix hands know YACC (now Bison) and Lex (now Flex).

   However creating a full-blown language is a lot of work: you have to write a parser, code generator / interpreter and possibly a debugger, not to mention all the routine stuff that every language needs like variables, control structures and arithmetic types. An embedded DSL (eDSL) is basically a short cut if you can't afford to do that. Instead you write the domain-specific bits as a library in some more general purpose "host" language. The uncharitable might say that "eDSL" is just another name for "library module", and its true there is no formal dividing line. But in a well designed eDSL anything you might say in domain terms can be directly translated into code, and a domain expert (i.e. a non-programmer) can read the code and understand what it means in domain terms. With a bit of practice they can even write some code in it.

   This paper describes an eDSL for financial contracts built in Eiffel which worked exactly that way. It doesn't talk about "domain specific language" because the term hadn't been invented back then, but the software engineers defined classes for different types of contracts that the financial analysts could plug together to create pricing models. Its interesting to compare it with this paper about doing the same thing in Haskell.

   But eDSLs have problems. The resulting programs are often hard to debug because a bug in the application logic has to be debugged at the level of the host language; the debugger exposes all the private data structures, making it

hard for application programmers to connect what they see on the screen with the program logic. The structure of the host language also shows through, requiring application programmers to avoid using the eDSL functions with certain constructs in the host language.

This is where Haskell comes in. Haskell has three closely related advantages over other languages:

Monads. The biggest way that a host language messes up an eDSL is by imposing a flow of control model. For example, a top-down parser library is effectively an eDSL for parsing. Such a library can be written in just about any language. But if you want to implement backtracking then its up to the application programmer to make sure that any side effects in the abandoned parse are undone, because most host languages do not have backtracking built in (and even Prolog doesn't undo "assert" or "retract" when it backtracks). But the Parsec library in Haskell limits side effects to a single user-defined state type, and can therefore guarantee to unwind all side effects. More generally, a monad defines a model for flow of control and the propagation of side effects from one step to the next. Because Haskell lets you define your own monad, this frees the eDSL developer from the model that all impure languages have built in. The ultimate expression of this power is the Continuation monad, which allows you to define any control structure you can imagine.

Laziness. Haskell programmers can define infinite (or merely very large) data structures because at at any given point in the execution only the fragment being processed will actually be held in memory. This also frees up the eDSL developer from having to worry about the space required by the evaluation model. (update: this isn't actually true. As several people have pointed out, while laziness can turn O(n) space into O(1), it can also turn O(1) into O(n). So the developers do have to worry about memory, but lazy evaluation does give them more options for dealing with it.)

The type system allows very sophisticated constraints to be placed on the use of eDSL components and their relationships with other parts of the language. The Parsec library mentioned above is a simple example. All the library functions return something of type "Parser foo", so an action from any other monad (like an IO action that prints something out) is prohibited by the type system. Hence when the parser backtracks it only has to unwind its internal state, and not the rest of the universe.

There are other programming languages that are good for writing eDSLs, of course. Lisp and Scheme have callCC and macros, which together can cover a lot of the same ground. Paul Graham's famous "Beating the Averages" paper talks about using lots of macros, and together with his patent for continuation-based web serving it is pretty clear that what he and Robert Morris actually created was an eDSL for web applications, hosted in Lisp.

But I still think that Haskell has the edge. I'm aware of the Holy War between static and dynamic type systems, but if I you put a Haskell eDSL in front of a

domain expert then you only have to explain a compiler type mismatch message that points to the offending line. This is much easier to grasp than some strange behaviour at run time, especially if you have to explain how the evaluation model of your eDSL is mapped down to the host language. Non-programmers are not used to inferring dynamic behaviour from a static description, so anything that helps them out at compile time has to be a Good Thing. And its pretty useful for experienced coders too.

(Update: I should point out that monads can be done in any language with lambdas and closures, and this is pretty cool. But only in Haskell are they really a native idiom)