JFP **22** (3): 375–377, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000123 First published online 18 May 2012

Bookreview JFP: Domain-Specific Languages by Martin Fowler The Addison Wesley Signature Series

JURRIAAN HAGE
Department of Information and Computing Sciences, Utrecht University P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* J.Hage@uu.nl)

My main reason for wanting to read this book was to find out what a well-known publicist from the world of OO would have to say about the state of the art of domain specific languages (DSLs), in particular when it comes to type error feedback, functional programming, and the combination. As most readers will be aware, languages like Scheme and Haskell are very well suited to embed DSLs in: Scheme can be considered a core language to which new language facilities can be easily added by means of hygienic syntax macro's (Abelson *et al.* 1998), and there are so many papers on embedded DSLs in Haskell (Hudak, 1998), that any realistic selection would aggravate more people than I would please. Great was my disappointment when I read on page XXV that these topics were not discussed at all in the book. Although I can imagine that Fowler does not feel comfortable writing about subjects he is not sufficiently at home with, the question does arise whether the title of this book is sufficiently covered by its contents.

What topics does the book adddress then? In short, about 160 of the 575 pages of the book are devoted to a general introduction to the field of DSLs: to motivate the notion of DSLs, to set the scene and terminology, and to provide some real life examples so that everybody will have come across at least one of them in his/her everyday programming life. The remainder of the book is devoted to explaining how to implement/design many of the necessary ingredients for internal and external DSLs (Fowler uses the term internal for embedded DSLs). Examples of such patterns include semantic models, macro's, embedded interpretation, symbol tables, and method chaining. In this sense, the book is very much like Terence Parr's Language Implementation Patterns (Parr, 2009) (also reviewed in JFP (Hage, 2011)), although in Parr's case this is much clearer from the title. Similar to Parr, Fowler provides different patterns to solve the same problem, and also discusses when or when not to apply a given pattern.

This does not mean that Fowler does not have anything to add to the book of Parr. The focus of Fowler is much more on the implementation of concepts that are particularly useful in the context of DSLs. Since Parr consistently assumes the external approach, this is particularly evident for patterns that apply specifically to internal DSLs. Moreover, the book of Parr is strongly tied to Java and Antlr, whereas Fowler also provides code examples in C# and Ruby. It is the flexibility of a dynamic language like Ruby that allows DSL developers to create the necessary fluency of a DSL inside a general purpose language. I shall return to the use of dynamic languages later on.

The parts most interesting to me are located in the first 160 pages. According to Fowler, the defining characteristics for a DSL are: programs written in a DSL must be executable, the language should have a sense of fluency (that may be particular to that domain), it should have limited expressiveness, and, largely as a consequence of this limited expressiveness, a domain focus. To me, the main advantage of a DSL is the ability to phrase solutions to domain problems in a more succinct and understandable fashion than is possible in general purpose languages; the critical ingredient is then for the DSL to have a fluency that seems natural to domain experts, and the ability to program at a sufficiently high level. However, if that can be arrived at without limiting expressiveness, I certainly do not want to blame a DSL for being Turing complete.

And what does limited expressiveness mean in the context of an internal DSL? The suggestion of Fowler (page 28 and 30) is that within the "fluency of the DSL" expressive power should be restricted. But I wonder if that is how a programmer will see it. Programmers make mistakes. It is all too easy to accidentally escape from the DSL into the host languages, particularly if the programmer is as uninformed about programming as we are led to believe. And what if the fluency of the general purpose language is not that different from that of the domain? Is that a drawback (because the distinctions become blurred) or an advantage (because general purpose programmers will have an easier time learning and remembering the DSL)? And what if DSLs are composed, should they be easily distinguishable from each other, or is similarity an advantage? Although Fowler does spend some time on such issues in Chapter 6, I cannot find satisfying answers to these general questions, outside the OO and imperative programming domain. As a result I am constantly thinking: does this argument apply to DSLs in the functional programming world as well?

When it comes to error diagnosis, Fowler admits to not addressing the issue sufficiently, although he certainly would have wanted to. According to him, bad error diagnosis is something that people will tolerate, and indeed there is a whole history of compiler builders not spending time on error diagnosis. Interestingly, in functional programming there has been quite a lot of work into improving type error diagnosis (see Heeren, 2005) for an overview until 2004, including work on domain specific type error diagnosis for Haskell 98 as implemented in the Helium compiler (Heeren *et al.*, 2003a,b). The complexity of type systems comes with a price: having parametric polymorphism and first class functions (and much much more), there are many more ways in which you can screw up your program. As such features find their way into more languages such as Java, either programmers will avoid fully exploiting these new features, or error diagnosis will become an issue. Coming back to the subject of DSLs there is something that does not feel quite right: on the hand the expressiveness of DSLs is limited to help non-programmers program within their limited domain, while on the other hand error diagnosis, of which such programmers have a dire need, is neglected as a side issue. Fowler's hope (page 63) is that DSL programs tend to be small anyway and finding bugs should not be too difficult. On the other hand, Chet Murty explained in his invited talk at POPL 2007 in Nice that his job is to rid his customers of millions of lines of XSLT, a language that Fowler, to some extent, will agree is domain specific.

To summarize: although the title does not make this clear, the focus of this book is strongly on OO languages, even if some of the concepts, e.g., nested closures, have a definite functional ring to them. If you happen to be interested in learning to construct tooling for DSLs, then the patterns described in this book can certainly be helpful, but remember that all the code you will see will be in Ruby, C# or Java. Conceptually, I am grateful to the book for stressing the idea of fluency, and putting together a (first?) attempt at a coherent picture of the world of DSLs. The fact that work done in the functional programming community is largely ignored, however, makes that I cannot recommend this book without reservations.

References

Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., Adams, N. I., Friedman, D. P., Kohlbecker, E., Steele, G. L., Bartley, D. H. & Halstead, R. (1998) Revised^5 report on the algorithmic language scheme. *Higher-order Symb. Comput.* **11**(1), 7–105,

Hage, J. (2011) Language implementation patterns: Create your own domain-specific and general programming languages, by terence parr, pragmatic bookshelf, http://www.pragprog.com, isbn 9781934356456. *J. Funct. Program.* **21**(2), 215–217,

Heeren, B. (2005) *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, http://www.cs.uu.nl/people/bastiaan/phdthesis.

Heeren, B., Hage, J. & Swierstra, S. D. (2003a) Scripting the type inference process. In *Proceedings of the Eighth International Conference on Functional Programming*. New York: ACM Press, pp. 3–13.

Heeren, B., Leijen, D. & van IJzendoorn, A. (2003b) Helium, for learning Haskell. In *Proceedings of the ACM Sigplan 2003 Haskell Workshop*. New York: ACM Press, pp. 62–71.

Hudak, P. (1996) Building domain-specific embedded languages. *ACM Comput. Surv.* **28**.

Parr, T. (2009) *Language Implementation Patterns*. The Pragmatic Bookshelf. http://www.pragprog.com/titles/tpdsl/language-implementationpatterns. Accessed 25 April 2012.