Construct Your Own Favorite Programming Language

S. Doaitse Swierstra

Technical Report UU-CS-2009-029
January 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Construct Your Own Favorite Programming Language*

S. Doaitse Swierstra

December 3, 2009

> Advanced PL technology is a secret weapon in enterprise computing
>
> Chet Murthy, POPL 2007

Abstract

We investigate how the need for evermore programming languages arises, and how to fulfill this need. We show how the gap between building a library using an existing language and constructing a completely new language narrows. In doing so we discuss a few of the past and current research results from the Software Technology group at Utrecht University.

1 Introduction

Almost as long as we have computers we have programming languages; starting from FORTRAN in 1958 a whole genealogy of language designs can be constructed[1] and one may wonder why computer scientists are not making up their mind, once and for all, and define the definite language which unifies all good ideas thought up thus far.

In this paper we both investigate where this urge for ever new languages stems from, and show how we might slow down the speed at which new languages are being proposed, defined, implemented, taught, and learned, only to be finally discarded and replaced by successors.

We start out by observing that a new language in principle does not add anything to what we can express already (e.g., using plain machine code); it may only, albeit dramatically, improve the ease with which we can express ourselves. Sometimes this new

*This paper has appeared in the book which was published on the occasion of 25 years of Computer Science education at Utrecht University

[1]For a large overview to put on the wall of your office see: `http://www.levenez.com/lang/`

expressiveness is so huge that we speak of a new paradigm; examples of languages introducing such new paradigms are Prolog (logic programming), Simula67 (object-oriented Programming) , Lisp (functional programming), ML (type inferencing) and SASL (lazy evaluation). In this paper we will not focus on such paradigm shifts, but also look at the more mundane evolution.

In Section 2 we discuss (an aspect) of the role computer science has in relation to other disciplines. It concludes with identifying an important cause for the wealth of languages we see around us. In Section 3 we introduce the two extreme approaches to the implementation of a new language: constructing a special purpose library or building a new compiler. In Sections 3.1 and 3.2 we discuss some developments in these two areas. Finally we mention some related research to which the Software Technology group has been contributing.

2 The role of computer science in relation to other disciplines

In the historical development of some area of knowledge we can distinguish a number of phases, for each of which we give a short characterisation:

amateur things are constructed without having real insight in how to proceed; cost, time, and quality are hard to judge beforehand; the wheel is reinvented.

craftsmanship artefacts are constructed on a routine basis; cost and quality are more or less predictable, but products do not vary a lot. Development is by trial and error.

scientific patterns are discovered in experimental results and theory is developed; mathematics is helping us to formulate basic principles and laws. Skilled people, who understand the formulae, can predict the quality of products. We can reason about why things do or will not work out as desired.

engineering unfortunately the formulated laws may be too complicated to be applied as they stand, so we develop procedures about when to apply formulae, and how to apply them. People who do not fully understand the formulae can be trained to apply them correctly: we can calculate, without having to interpret intermediate formulae. Cost becomes predictable.

automation by the time we fully understand the underlying principles, where to apply them and in what order, we can fully specify what we want to construct, and can leave it to a machine to do the actual construction. Computer science has a similar rôle here as mathematics has in the third phase.

Those who know the Capability Maturity Model, which was developed by the Software Engineering Institute, will have no problem seeing that there is a close correspondence between the above-mentioned phases and the levels of maturity as described by the CMM model.

In order to be able to get to the final stage we thus need a complete formal description of an area of knowledge; unfortunately a complete description is not always available, so we see two tasks for computer scientists here:

1. provide assistance to other areas of knowledge in formulating their concepts, laws, invariants, processes, such that the descriptions made by the experts in that specific area become executable on a computer;

2. provide the mappings from these formulations to executable programs.

One might call the result of the first step a *domain-specific programming language* (DSL), since it is specific for the domain at hand. We conclude that the large increase of the number of such languages is a sign of more and more disciplines progressing towards the *automation* stage; each area of knowledge has its own terminology, ways of formulating things, be it by drawing diagrams, using formulae, or using formalised language.

So do we have a problem? The answer is definitely affirmative. Designing some domain-specific language may initially seem simple, and a first step may already provide a very successful product, since its use cuts down on coding time, avoids common errors, and makes theory widely available. Unfortunately this first step is often not the last one and there are many more to follow, driven by the success of the first one. Once we have the initial design for a DSL, we start to use it and as a result get inspiration about how to improve it, and something which looked simple and elegant at the beginning soon grows into a monstrous and inconsistent design, "leaving it up to the customer to find out which combination of features work".[2]

How to make the transition into the fifth stage is itself again subject to the mentioned staging and is often done in an amateur way, without the eye of a trained computer scientist being around to help and warn.

As a consequence the world abounds with ill-designed programming languages, which typically exhibit one or more of the following problems:

- *incomplete* One may start out with a concept A, only to discover later that concept B is also needed, and so this is added; by the time one adds concept C the insight comes that all are actually instances of a more abstract concept X, i.e. $A = X(a)$, $B = X(b)$ and $C = X(c)$. By this time it may be too late however to add the more abstract concept X to the language in a proper way, since its introduction interferes with other constructs which are not easily modified. As a result one decides to proceed without properly introducing X, and then forgets to provide the instances of X for other parameters, such as d. A typical example of this is the possibility to read and write (serialise and deserialise) values; not all types of values are treated equal here.

- *inconsistent* It may be the case that the concepts A and B, as introduced before, are actually instances of slightly X' and X'', where X and X' only differ slightly. By the

[2]quoting Charles Simonyi, the lead developer of MS Office, advocate of *intentional programming*, inventor of the Hungarian notation and space tourist.

time the insight is there it may be too late to introduce the proper concept X, since a lot of already constructed software depends on the slight difference in semantics between X' and X'' . This sometimes described by: "Every well documented bug becomes a feature".

- *error prone* Making the compiler check whether a specific program makes sense at all, is a lot of extra work, so many DSLs are de facto dynamically typed, just as most scripting languages. From the language implementers point of view this is optimal, since he does not have to pay attention to this aspect. Some programmers also see it as a feature that the compiler never complains about an inconsistent program. The fact that, if the language turns out to be successful, large numbers of programmers will be spending a lot of time finding errors which could have been detected automatically, is something one only regrets once it happens. We argue that any conceivable static check that helps in detecting errors should be applied.

- *lack of abstraction mechanisms* Many language designs start out with no abstraction mechanisms, and once the need arises some form of macro mechanism is added; sometimes by just calling the C-preprocessor. This unfortunately does not always work well, and may prohibit static checking.

- *semantic lock-up* In defining the semantics of the language we may not spend enough time to important details, which turn out to become problems later. A nice example is the absence of automatic garbage collection in languages as C. The designers wanted to keep things simple and judged that garbage collection was something which was best left to the programmer, since with his intimate knowledge of the program he knew best when and where to free parts of memory. Once this is a widely used practice, combined with the possibility to mix pointer arithmetic with integer arithmetic, there is no way back. The amount of money lost in programming time, debugging time, and recovering from failing programs is too large to be even estimated. Quite a number of security violations stem from uncontrolled buffer overflows, which cannot be statically avoided by a proper program analysis; since the languages were never designed to be subjected to such analyses the programs themselves have become time bombs.

3 How to realise a language implementation

As we have argued before there is the one hand a good reason for having quite a few domain-specific languages, but on the other hand we have seen that defining a complete language is not an easy task, and so the question arises *How to Support the Design and Implementation of Domain-Specific Languages?* We can attack this problem from two sides: *library construction* and *compiler construction*.

From the very beginning general-purpose programming languages have had procedures and functions, and early on it was recognised that this would enable the construction

of expressive libraries; for example, a fully developed numerical library can be seen as a domain-specific language, built from concepts from linear algebra and analysis. Although the availability of such a "library language" formally does not change the language a program is written in, the user is actually programming using the concepts from the library. In such cases we speak about an *Embedded Domain-Specific Language*. Given the success of this approach the question arises: *Which features of a host language make it a good vehicle for embedding other languages?*

At the other end of the spectrum we find the newly designed languages, an approach which has become popular because tools for the construction of compilers (parsers, lexers, attribute grammar systems) have become widely available. Since languages are often not so different we might construct a library of programming language concepts, from which we can compose new languages easily.

In the next two subsections we will discuss each of these approaches a bit further.

3.1 Easier and better ways of building libraries

It is our claim that modern, polymorphically typed, lazy functional programming languages such as Haskell [14], are currently the most appropriate vehicle to embed other languages in.

minimalistic The designers of Haskell have tried to keep the language design as small as possible; so there is e.g. no distinction between an expression and a statement: the only thing which can be computed is an expression. This makes that we also need only a single abstraction mechanism for procedures and functions.

full abstraction Most special-purpose programming languages have poorly defined abstraction mechanisms, often not going far beyond a simple macro-processing system. Although –with a substantial effort– amazing things can be achieved in this way as we can see from the use of TEX, we do not think this is the right way to go; programs become harder to get correct, and often long detours –which have little to do with the actual problem at hand– have to be taken in order to get things into acceptable shape. Because our embedded language inherits its abstraction mechanism from Haskell –by virtue of being an embedded language– it takes a head start with respect to all the individual implementation efforts.

type checking Most DSLs, and especially the so-called scripting languages, only have a weak concept of a type system. Haskell, however, has a very powerful type system, which is not easy to surpass, unless one is prepared to enter completely new grounds, as with dependently typed languages such as Agda [12]. One of the huge benefits of working with a strongly typed language is furthermore that the types of the library functions already give a very good insight in the role of the parameters and what a function is computing. The polymorphic data types (i.e., data types with type parameters) enable us to define new type constructors, and thus new ways of combining already existing types into new types.

clean semantics One of the ways in which the meaning of a language construct is traditionally defined is by its denotational semantics, i.e., by mapping the language construct onto a mathematical object, usually being a function. This fits very well with the embedding of domain-specific languages in Haskell, since functions are just normal values in Haskell. As a result, implementing a DSL in Haskell almost boils down to giving its denotational semantics in the conventional way and getting a compiler for free.

lazy evaluation One of the formalisms of choice in implementing the context sensitive aspects of a language is by using attribute grammars. Fortunately, the equivalent of attribute grammars can be implemented straightforwardly in a lazily evaluated functional language; inherited attributes become parameters and synthesised attributes become part of the result of the functions giving the semantics of a construct [19, 20].

embedding syntax A new language often comes with new notation. In Haskell such new notation can be mimicked by introducing new operators with appropriate precedences and associativities. One of the prime examples here are the so-called parser combinators, which make it possible to write down expressions which closely resemble context-free grammars, but effectively evaluate to parsers. Using the class system in a very cunning way we can even deceive the Haskell compiler to handle completely new notation [11, 18], which is still an expression but no longer easily recognised as such.

Of course there are also downsides to the embedding approach. Although the programmer may think he writes a program in the embedded language, he is still programming in the host language. As a result of this, error messages from the type system, which can already be quite challenging in Haskell, are phrased in terms of the host language constructs too, and without further measures the underlying implementation shines through. In the case of our parser combinators, this has as a consequence that the user is not addressed in terms of terminals, non-terminals, keywords, and productions, but in terms of the types implementing these constructs.

This problem has been addressed by the thesis of Heeren [5, 7]: the Haskell type checker is extended in such a way that generated error messages can be tailored by the programmer. Now, the library designer not only designs his library, but *also the domain-specific error messages* that come with the library. In the Helium compiler [6, 8], which handles a subset of Haskell, this approach has been implemented with good results. An example of what can be achieved is given in figure Fig. 1, which contains a program which uses the parser combinators in an incorrect way. In Fig. 2 we see that Helium, by using a specialised version of the type rules –which are provided by the programmer of the library–, manages to address the application programmer in terms of the embedded language; it uses the word *parser* and explains that the types do not match, i.e. that a component is missing in one of the alternatives. A final option in the Helium compiler is the possibility to program the search for possible corrections, e.g. by listing functions which are likely to be confused by the programmer (such as <*> and <* in programming parsers, or : and ++ by beginning

```
data Expr     = Lambda Patterns Expr
              -- can contain more alternatives
type Patterns = [Pattern]
type Pattern  = String

pExpr :: Parser Token Expr
pExpr
  =  pAndPrioExpr
 <|> Lambda <$  pSyms "\\"
            <*> many  pVarid
            <*  pSyms "->"
            <*  pExpr         -- <* should be <*>
```

Figure 1: Type incorrect program

```
Compiling Example.hs
(7,6): The result types of the parsers in the operands
       of <|> don't match
  left parser   : pAndPrioExpr
    result type : Expr
  right parser  : Lambda <$ pSyms "\\" <*> many pVarid
                                       <* pSyms "->"
                                       <* pExpr
    result type : Expr -> Expr
```

Figure 2: Helium, version 1.1 (type rules extension)

Haskell programmers), and to see whether parameters have been swapped accidentally. This will help in learning the embedded language. As we can see in Fig. 3 we can now pinpoint the location of the mistake even better and suggest corrective actions.

One important development to which Utrecht has contributed is the phased implementation of embedded languages, thus using techniques from conventional compiler construction in the implementation of embedded languages: the embedded program is first represented as a *typed abstract syntax tree*, which is subsequently analysed and from which finally a function representing its semantics is constructed. In this way a very efficient, error correcting parser combinator library has been produced [17], which has served as one of the primary examples of this approach, and became the inspiration for the introduction of *arrows* [9] into Haskell.

Haskell has now been used for embedding a wide variety of languages, e.g., by Leijen [10] for accessing databases in a typed way, thus providing a safer and less cumbersome alternative to embedded SQL-queries. For an overview one may consult the Haskell web-

```
Compiling Example.hs
(11,13): Type error in the operator <*
  probable fix: use <*> instead
```

Figure 3: Helium, version 1.1 (type rules extension and sibling functions)

site[3].

As mentioned before an important aspect of embedding a language is that the embedded language inherits its type system from the host language, and thus we have to make sure that the types are also reflected in the abstract syntax trees representing the embedded programs. It is remarkable that this can be done in Haskell itself [1]. This patterns has become so common that it has led to the introduction of *generalised algebraic data types* (GADTs) into Haskell, of which we will give a small example.

One may represent expressions, with the types of the values represented by the expressions as type parameters, as follows:

data $Expr\ a$ **where**
$$\begin{aligned} &Val &&:: a \to Expr\ a \\ &Apply &&:: Expr\ (b \to a) \to Expr\ b \to Expr\ a \end{aligned}$$
$$\begin{aligned} eval\ (Val\ a) &= a \\ eval\ (Apply\ tf\ ta) &= (eval\ tf)\ (eval\ ta) \end{aligned}$$

Now suppose want to add a polymorphic *Pair* constructor to the language, which takes two values of possibly different types, let us say a and b. Then the value which is represented has type (a, b). Unfortunately we have no easy way to represent this type using the normal Haskell data types. Using the GADT extension however we can now add an extra constructor representing pairs:

data $Expr\ a$ **where**
$$\begin{aligned} &... && \text{-- as before} \\ &Pair &&:: Expr\ a \to Expr\ b \to Expr\ (a, b) \end{aligned}$$

The function *eval* also gets an extra alternative:

$$eval\ (Pair\ ta\ tb) = (eval\ ta, eval\ tb)$$

The function *eval* can now conclude from its argument being a *Pair* that that the *Expr* passed was indeed labelled with a pair type.

Using such algebraic data types we have developed a library which makes it possible to manipulate terms which represent declarative structures (such as grammars, or a set of mutually recursive definitions) in a type-safe way [2]. This library was used in building efficient *read* functions. For every data type we want to serialise we generate a typed data structure describing its "grammar". When the modules are linked these grammar fragments are dynamically combined, analysed and transformed, and finally mapped onto a parser. This can all be done in a type-safe way; thus one gets a partial correctness proof of the finally constructed function for free [21].

[3] `http://www.haskell.org/haskellwiki/Applications_and_libraries`

3.2 Composing languages and their implementation

Starting from the other end we may try to build libraries which contain "language fragments", and libraries which contain common analyses. Out of these one can quickly construct a compiler for some domain-specific language.

The approach we take here is to lean heavily on attribute grammars (a domain-specific language for compiler construction), which allow us to describe language fragments in isolation, and to easily combine them in constructing a compiler. We use this technique extensively in the construction of our own Haskell compiler [3], the details of which can be found on its website `http://www.cs.uu.nl/wiki/bin/view/Ehc/`. This compiler is developed along two axes: along one axis we incrementally build the language which is compiled, and along the other axis we incrementally add the aspects a compiler has to deal with, such as parsing, type checking, error reporting, program analysis, optimisation, and code generation.

Despite being small, Haskell is a large language with a complicated type system, which allows to infer most of the types; only higher ranked types, existential types, GADTs, classes and polymorphic recursion need explicit type annotations. In order to be able to experiment easily with different versions of the type system we have designed yet another domain-specific language *Ruler* [4] for describing type systems; from such descriptions we generate attribute grammar based implementations, provided the definitions have a specific shape. One of the areas of research is to liberate the current restrictions on the shape, and to be able to handle more declarative formulations fully automatically.

One of the observations which can be made is that in the Haskell world the development of the language and its compilers, and the development of the libraries goes hand in hand. It is our view that in the future it will be possible to build libraries and compilers within a single unified framework, where languages can be easily extended, the properties of such extensions can be easily specified, and programs written using such extensions can be easily verified.

4 Supporting programming

Most DSLs still look like a normal programming language, i.e., they require a linear, character based representation of their programs. People working with model-based development are however used to more graphical representations (such as UML), and thus the question arises whether we can provide more domain-specific representations. In order to make the use of DSLs easier we have developed the Proxima framework [16], which provides much more liberty in presenting and editing programs. In Fig. 4 we show the representation of a Helium program, in which formulae are presented as they would occur in a mathematics book, instead of as they are usually represented in a programming language and thus "FORTRAN (Formula Translator) is finally there". Note that the type information given here is all computed by the combination of the editor with the compiler, such that the programmer gets automatic feedback when constructing his program.

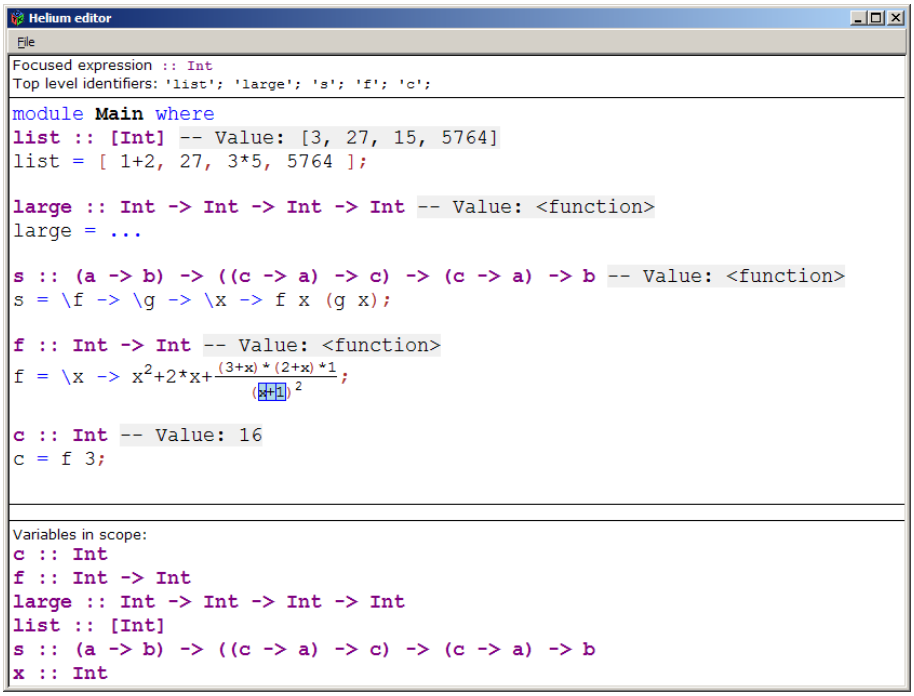

Figure 4: The Helium editor at work

A final observation which can be made is that with the gradual merging of type checking, program verification, providing editing feedback and code generation, the need for incrementally evaluated systems grows; analysing large artefacts over and over again becomes infeasible. In Utrecht the work of Vogt [23], Pennings [13] and Saraiva [15] resulted in an incrementally evaluated attribute grammar system for higher order attribute grammars [22]. We plan to move these results over into our Haskell based tool set in the near future, so they become automatically available in the Proxima framework, which uses these tools heavily.

References

[1] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002.

[2] I. Baars, Arthur, Doaitse Swierstra, S., and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

[3] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. In *Implementation and Appli-*

cation of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, volume 5083, pages 57–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006*, number 3945 in LNCS, pages 30–46. Springer-Verlag, 2006.

[5] Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, 2005.

[6] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.

[7] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[8] Bastiaan Heeren and Daan Leijen. Functioneel programmeren met Helium. In *NIOC'2004*, pages 73–82, November 2004.

[9] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[10] Daan Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, November 2003.

[11] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.

[12] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[13] Maarten Christiaan Pennings. *Generating Incremental Attribute Evaluators*. PhD thesis, Utrecht University, Department of Computer Science, Padualaan 14, 3508 TB Utrecht, November 1994.

[14] Simon Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1):7–255, 2003.

[15] João A. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, December 1999.

[16] Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, Oct 2004.

[17] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

[18] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical Report UU-CS-2008-044, Department of Information and Computing Sciences, Utrecht University, 2008.

[19] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João A. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[20] S. Doaitse Swierstra, Arthur Baars, Andres Löh, and Arie Middelkoop. UUAG-Utrecht University Attribute Grammar System.

[21] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell Symposium*, pages 63–74, New York, NY, USA, 2008. ACM.

[22] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Sigplan 1989 Conference on programming language design and Implementation*. ACM, 1989.

[23] Harald Heinz Vogt. *Higher Order Attribute Grammars*. PhD thesis, Utrecht University, Department of Computer Science, Padualaan 14, 3508 TB Utrecht, Feb 1993.