DOMain Specific Type Error Diagnosis (DOMSTED)

Jurriaan Hage

Technical Report UU-CS-2014-019
July 2014

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

DOMain Specific Type Error Diagnosis (Project Paper)

Jurriaan Hage

Department of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
J.Hage@uu.nl

Abstract. Domain-specific languages (DSLs) have the potential both to reduce the effort of programming, and to result in programs that are easier to understand and maintain. For various good reasons, researchers have proposed to embed DSLs (then called EDSLs) into a general purpose host language. An important disadvantage of such an embedding is that it is very hard to make type error diagnosis domain-aware, because inconsistencies are by default explained in terms of the host language. In earlier work we have developed a method to make type error diagnosis domain-specific, and we have applied the method to Haskell 98.
The practice of Haskell programming shows that currently applications and libraries employ type system features well beyond those offered by Haskell 98. Here lie both a practical and fundamental challenge that the project aims to address. It is practical because only after meeting this challenge, can our ideas be employed in everyday programming; it is fundamental because an essential understanding of Haskell's type system features, such as GADTs and type families, is necessary to achieve the necessary control over the type system that can then be passed on to the EDSL developer.
Our work will enable EDSL developers to safely, transparently and non-invasively provide domain specific type error diagnosis. It is transparent because the designer of the rules does not need to have intimate knowledge of the internals of the compiler, safe because it cannot be used to circumvent the strong type system, and non-invasive since the EDSL code itself need not be changed.
The work in this project will be undertaken by a PhD student under the supervision of the first author of this paper, in collaboration with Atze Dijkstra and others at Utrecht University.

Keywords: type systems, type error diagnosis, embedded domain specific languages, Haskell

1 Introduction

Over the last few years, many computer scientists and professionals have shifted their attention from general purpose languages (GPL) to *domain specific languages (DSLs)* [22, 20, 16], having realized it is much more productive to program

in languages designed for effectively solving problems in a particular domain. Moreover, a DSL is often easier to grasp by non-professionals and therefore can expect to have more wide-spread use. A well-known example is the SQL language for database querying.

According to Walid Taha [31], a DSL has four essential characteristics: the domain is well-defined and central, the notation is clear, the informal meaning is clear, and the formal meaning is clear and implemented. It is the latter characteristic that sets a DSL aside from a jargon [31]. To these characteristics, we add:

- an implementation of the DSL can communicate with the programmer about the program in terms of the domain.

The rationale is that programmers make mistakes, and communicating the diagnosis of a mistake in terms other than that of the domain completely defeats the purpose of working with a DSL!

Although providing domain level feedback can demand a substantial engineering effort, the problem is not particularly difficult if for every DSL we essentially implement a new compiler. But this is not always the best approach. In [21], Paul Hudak argues that DSLs are the "ultimate abstraction", and introduces the idea of embedded DSLs (EDSLs) (also called internal DSLs [8]). EDSLs typically inherit the style, syntax, type system, infrastructure, and tooling of a chosen host language. There are significant advantages:

- The complete infrastructure of the host language can be reused, e.g., libraries, code generation, debuggers, implementation of floating point numbers, that are costly to implement and maintain for separate DSLs.
- EDSLs (for the same host language) can typically be combined relatively easily. Combining multiple DSLs, each with their own tool chain, is a daunting task by itself and at worst for every useful combination a separate tool chain must be maintained.

Arguably, a further benefit is that when the EDSL is not expressive enough to solve a particular problem, the programmer can always fall back on the host language, which is usually Turing complete. Finally, the host language provides a form of basic syntax, leading to EDSLs that are similarly styled and therefore may be easier to learn.

EDSLs also have two important disadvantages: since the EDSL is embedded inside (encoded into) some general purpose language, the compiler for the latter has no awareness of concepts in the EDSL, which leads to the inabilities

- to report type error messages in terms of the domain (domain specific error diagnosis),
- to exploit knowledge of the domain to generate better code (domain specific optimisations).

The DOMSTED project aims to address the first of these two issues: domain specific type error diagnosis. We shall do so within the context of the Haskell programming language [9].

In earlier work [19], we have shown an approach to making type error diagnosis domain-specific for Haskell 98 (a subset of Haskell). Because of its importance for this paper, we introduce the specification language (a DSL!) for *specialized type rules* that it describes.

As an example, consider a parser combinator library, i.e., an extensible implementation of EBNF. The primitives in such a language are parsers that: always succeed (*succeed*), always fail (*fail*), and recognize a symbol (*symbol*). Moreover, there are combinators that construct parsers out of other parsers: for parsers *p1* and *p2*, *p1* <∗> *p2* parses *p1* followed by *p2*, *p1* <|> *p2* parses *p1* or *p2*, and *f* <$> *p1* parses *p1* and transforms the outcome by the function *f*. In the example below, we use the *option* combinator that takes a value x of type a, and a parser p that produces values of type a, so that *option* p x is the parser that parses zero or one p, returns x if p fails, and the result of the parse if it succeeds. With these in hand, we can define a parser that given a parser p and a number n constructs a parser that parses at most n instances of p and returns the results in a list, as follows.

atmost p 0 = *succeed* [ ]
atmost p n = *option* ((:) <$> p <∗> *atmost* p $(n-1)$) [ ]

This implementation is type correct, but what happens if the arguments to *option* are interchanged? A domain-agnostic type error message may then expose the underlying implementation of parsers to the user, as the following message (from Hugs) shows:

```
ERROR "Parsex.hs":17 - Type error in application
*** Expression     : option [] ((:) <$> p <*> atmost p (n - 1))
*** Term           : []
*** Type           : [a]
*** Does not match : [b] -> [([c] -> [([d],[c])],[b])]
```

To make the message domain aware, so that the error messages themselves can abstract away from underlying details, we have defined a specialized type rule for the `option` parser in Figure 1. The rule consists of a type rule, at the top, followed by a list of constraints. The expression below the line `option p v` tells us to which kind of expression the rule applies. Because `p` and `v` occur above the line, these variables are considered to be meta-variables that range over all Haskell expressions. The identifier `option` does not occur above the line, which tells us that this is a particular identifier that is in scope as the rule is read by the compiler. In other words: this rule applies to all calls to the function `option` irrespective of the form of its two arguments.

Behind each of the occurences of `::` the rule has a type associated with the expression before the `::`. In this case `p` and `v` have a variable type, which by itself imposes no restriction on them, while the type of the consequent is `Parser t6 t4`; this tells us that the result type is a parser of some sort (*t6* and *t4* are at this point still unconstrained).

```
    p :: t1 ;    v :: t3 ;
------------------------------
    option p v :: Parser t6 t4 ;;

t1 == Parser t5 t2: "@expr.range@: @p.pp@ should be a parser"
t5 == t6: "@expr.range@: the parsers @p.pp@ and @expr.pp@
          "    do not work on the same kind of token"
t2 == t4: "@expr.range@: the return type of @expr.pp@ and
          "    @p.pp@ do not coincide"
t2 == t3: "@expr.range@: the type @t3@ of the optional value"
          "    @v.pp@"
          "does not match the parser return type @t2@";
```

Fig. 1. A specialized type rule for *option* parsers

Further restrictions are imposed by the constraints below the rule, such as `t1 == Parser t5 t2`. The constraints are a convenient syntax for the unifications that need to be performed to verify the type correctness of the consequent, `option p v`.

The constraints below the rule are, during type inference, checked from top to bottom. This means that as we check that `t5 == t6` (i.e., we unify the type associated with `t5` with the one associated with `t6`), we know that the previous constraint(s) could be satisfied. We can employ that information in the type error message that follows the constraint. Because the first constraint tells us that `p1` is a parser of some kind, we do not run any risk when saying so in the error message that comes with `t5 == t6`. Although our work allows a larger variety of control over the way constraints are solved (unifications are performed), this example captures the essence of our work.

The Helium compiler [15] implements these ideas and generates the following two type error messages (because as it happens, there are two unsatisfiable constraints, `t1 == Parser t5 t2` and `t2 == t3`):

```
(17,14): [] should be a parser
(17,14): the type Parser a [b] of the optional value
            (:) <$> p <*> atmost p (n - 1)
         does not match the parser return type [b]
```

It is important to realize that a compiler that supports specialized type rules does not simply generate good type error messages: it only provides the means to DSL developers to control type error diagnosis. In other words, we provide a mechanism or infrastructure that can be employed by DSL developers to proscribe type inferencing policies. Such policies may or may not lead to improvements in type error diagnosis, depending on whether the policies actually make sense. To warn against nonsensical modifications, however, the specialized type rules are automatically checked for soundness with respect to the built-in type system. More aspects of specialized type rules are discussed in [19].

In 2003, Haskell 98 was the standard, although GHC already supported many extensions that our work did not yet support. Since then many new developments have taken place within the Haskell community: new type system concepts have been developed and experimented with and, as we show in the next section, are used by many for the development of libraries and applications. Moreover, type system concepts such as GADTs that have been in flux for some years, now seem to have stabilized, making that this is an excellent time for undertaking the work described in this paper, which is to scale the ideas of [19] up to a full and mature language.

We have extracted from the Hackage [1] database which extensions are enabled for then most recent version of all available packages (Hackage contents of June 4, 2012, 4164 packages with a total of 5501 executable and library sections). The extensions that affect the underlying type system and that are enabled most often are listed in Figure 2. For each such extension we give, in the second column, the number of packages that explicitly document in their cabal file that the package employs that extension. The final column of Figure 2 contains the number of packages of the Top 20 downloaded libraries from Hackage from 2010 [28] that either by their cabal file, or their source code, document the use of the extension. The number excludes the extensions needed for compiling any dependencies. While doing this we learnt that the extensions listed in the cabal files are, as a rule, only a small subset of the extensions that are listed within the source code itself. In other words, the numbers in the second column are in reality probably much higher than listed here.

Extension	#enabled on Hackage	# enabled among library Top 20
FlexibleInstances	332	10
MultiParamTypeClasses	321	9
FlexibleContexts	232	3
ScopedTypeVariables	192	3
ExistentialQuantification	149	6
FunctionalDependencies	139	4
TypeFamilies	114	1
OverlappingInstances	108	3
Rank2Types	100	3
GADTs	88	3
RankNTypes	81	1
UnboxedTuples	20	4
KindSignatures	20	0

Fig. 2. Extensions enabled on Hackage

Haskell has many other extensions besides those mentioned in Figure 2, e.g., *TypeSynonymInstances*, *Arrows*, and *TypeOperators*, but these extensions do

[1] The infrastructure through which Haskell programmers share libraries and applications.

not complicate its type system. Dealing with these extensions is therefore expected to be largely a matter of additional engineering effort, and not to demand additional research. We did find during our experiments that certain extensions are employed quite often, so in order for our work to be used extensively, we cannot simply ignore them.

2 Research questions addressed within this project

The research question central to the project is whether we can control the type inference mechanism of Haskell with its many advanced type system features to the extent that we can exploit that control to cater for domain-specific type error diagnosis.

We aim to prototype our work in the locally developed Utrecht Haskell Compiler [3]. It may be possible that the prospective PhD student spends time as an intern at Microsoft Research in Cambridge in order to implement his work into the GHC. This internship shall be in addition to the four year appointment of the PhD. The internship ensures a degree of research impact, while allowing the student to focus on scientific issues during the PhD period.

Validation of our work amounts to developing specialized type rules for Haskell EDSLs and showing that they behave as they should for a substantial number of applications and libraries that employ the type system features under our control. We shall not study whether having domain specific error messages, as compared to generic ones, are profitable for the Haskell programmer. We leave it to developers of EDSLs to judiciously employ our work.

Our work can seriously impact the practical application of EDSLs, at first particulary within the Haskell community. To achieve this, we have to arrive at a fundamental understanding of a large variety of type system concepts and their interactions, because the envisioned control over the type inference process can only be achieved by a deep understanding of the language concepts that make up Haskell. A second fundamental challenge is to guarantee that modifications of the type system do not change the set of acceptable programs, and to do so automatically.

Our work can also serve as an enabler for language design. For example, some researchers advocate the reintroduction of monad comprehensions in Haskell [10]. According to Chakravarty [12], these were taken from the language, because type error messages for monad comprehensions were incomprehensible to people who were only familiar with list comprehensions. Within this project, we can investigate whether we can accommodate monad comprehensions generally by employing specialized type rules to specialize type error diagnosis for list comprehensions.

The control over the type system also naturally paves the way for the development of heuristics to improve type error diagnosis for Haskell, generally. Combinators for exercising such control over Haskell 98 are described in [14]. Heuristics for Haskell 98 can be found in [13] and [18]. Some of these heuristics also arise in work on security type error diagnosis [33].

3 Approach

The developers of GHC, the foremost compiler for Haskell [9], have added quite a number of language concepts to GHC over the years (before, in fact, a committee was re-established), and are at times quite active in the redesign of Haskell's type system and implementation. A recent such activity is documented in [32], and this document forms the starting point of our work.

In the case of Haskell, there is a large range of advanced type system concepts to choose from, including but not limited to: existential types, higher-ranked types, GADTs, type families, and multi-parameter type classes. As we showed earlier, these facilities are used often in everyday Haskell programming.

We advocate an incremental approach, starting out with those type system concepts that, judging from Figure 2, seem most useful, and formulate these type system concepts in terms of manipulable constraints. Recent redesigns of Haskell's type system and its implementation already employ constraints, which probably eases our task. At this time we also need to consider interference with already considered constructs, and extending the algorithm that checks that modifications to the type system do not inadvertently change the intrinsic type system. During the development, the results will be implemented in prototype fashion into the UHC [3].

Each type system concept, say GADTs, can be considered at increasing levels of maturity. For example, can we add support for GADTs

I so that existing type rules work for programs that contain GADTs as long as they do not match on expressions that contain GADTs,
II so that existing type rules work for programs in which meta-variables can be associated with expressions that contain GADTs, and
III so that specialized type rules can be written that match on GADTs.

We aim to allow up to level II in all cases, while level III will only be useful for certain type system concepts. For example, we currently cannot pattern match on lambda-abstractions, although meta-variables may be bound to such expressions. We do not yet know which constructs can profit from the level III treatment, but by considering a range of example EDSLs taken from Hackage, this can be discovered along the way.

Technical challenges are likely to arise in proving strong (automatically verified) guarantees that the specified modifications to the type system do not inadvertently change the type system itself. We have observed that writing down the specialized type rules in a way consistent with the intrinsic type system is not an easy task, and that this should be supported by automated soundness checks, and a diagnostic facility to explain what is wrong if inconsistencies arise. As the language grows in complexity, this facility will be in need of further maturation, and for the newly added type system concepts, this can be a particularly challenging aspect.

From a software engineering perspective, specialized type rules do not yet scale very well. First, there is a lack of abstraction facilities: each specialized

type rule essentially stands on its own. At the very least, some sort of macro facility will be necessary to standardize the type error format. Also, the rules lack facilities to tailor type error messages based on the computed types. For example, for the monad comprehensions mentioned earlier in this paper we want the type error messages to be specialized for certain instances of an expression, depending on the type of monad, e.g., the list monad. Such a facility calls for a conditional structure in the type rules that allow us to further specialize type error diagnosis. Furthermore, we also need to address the issue of composing collections of specialized type rules, because a single application may employ various EDSLs at the same time. We expect these issues to be addressed throughout the project.

Finally, we envisage an extension to specialized type rules to better control their application. As the expressions in the consequent of a specialized type rule become more complicated, and multiple EDSLs are applied within a single application, we should be able to describe strategies that govern how the type rules are to be employed.

4 Related Work

The field of domain specific languages is booming, and a huge number of publications exist. To provide some structure to the field, there have been a number of surveys, e.g., [2]. Hudak [21] was the first to propose the idea of embedded domain specific langage (EDSL), and he also discusses some of the essential ingredients: instrumentation, partial evaluation and modular, monadic interpreters. Swierstra [30] adds that DSLs should inherit the host language type system, which then needs to be stored in the abstract syntax trees that represent the embedded programs. To this end, the type system concept of Generalized Algebraic Data Type (GADT) was added to the Haskell language.

The Ph.D thesis of Bastiaan Heeren [17] contains a comprehensive overview of the field of type error diagnosis for functional languages up to 2004. Most of these papers discuss type error diagnosis for the polymorphic lambda-calculus, and do not discuss domain specific error diagnosis (except [19]).

As to languages beyond the typed lambda calculus, we are aware of the following: Rahli, Wells and Kammaredine scale the work on type error slicing up to a full-sized language, ML [27]; this work also shows that scaling up an idea from a core calculus to a full-fledged language is by no means trivial. There are few other attempts that go beyond the polymorphic lambda-calculus [1]. In [18], we considered type error diagnosis for Haskell's type classes, which are a means for specifying ad-hoc overloading, and form an instance of the theory of qualified types [23]. Some of these directives were developed independently by Wazny et al. [11], and are part of the Chameleon interactive type debugger. Furthermore, in his Ph.D thesis, Wazny also considers algebraic data types and explicit annotations. The work on Helium also accounts for these particular type system concepts [17].

Besides the work of the PI [19], there is little work on domain aware type error diagnosis. JavaCop [24] is a system that allows type system fragments to

be plugged into the Java type system, so that various kinds of properties can be checked at compile-time. Like [19] it is declarative, but since it uses data-flow support and Java does not support closures (yet), the technical differences are quite large. JavaCop provides a test harness to simplify the testing of the pluggable type systems. In our work this is unnecessary, because we can automatically prove the intrinsic type system is not changed. Our work, however, does not allow the type system to be changed at all, and JavaCop purposefully does. The pluggable type syntax has facilities to inform programmers when a program fails to compile.

5 Context and perspective

This project will be carried out within the Software Technology group at Universiteit Utrecht, which has a long history of building compiler oriented tools [7]. Work progresses on tools such as the UHC [4], many of which were built using locally developed tools that will all surely play a role in this project [29].

The influence of functional languages should not be underestimated. For example, parametric polymorphism and closures have both found their way into mainstream languages, and other functional languages such as OCaML, Scala, and F# are on the rise outside academic circles. This project allows us to experiment with many advanced type system concepts in the relatively clean context of Haskell that moreover can boast an active user community. Hackage will allow us to disseminate our work, and provides many examples of EDSL for treatment.

The project does not yet account for the complicating issue of subtyping; we save that for future proposals. We also do not expect to address Template Haskell in this project, at the very least saving that particularly complex extension for last. We have looked into controlling type error diagnosis for Generic Java [6, 5], but have not yet looked into the issue of specialized type rules in this setting. Indeed, we are more likely to set our sight on Scala [26] and Timber [25] for such an endeavour.

6 Project execution

Figure 2 gives an indication what we should address within this project. At the very least: rank-n types, GADTs, multi-parameter type classes, functional dependencies, type families, and existential quantification. However, each concept can be addressed up to various levels (I, II or III). Although it may not be the most effective to reconsider concepts at various stages of the research, we believe it is essential that all the important aspects are addressed up to level I as soon as possible, if possible up to level II. Having reached these levels will at least allow users to experiment soon with the technology. We can then selectively provide level III support for a growing range of concepts.

The order in which we shall address the extensions is also influenced by how complicated we believe the concept to be. For example, because we know how to deal with type classes [18], the step to multi-parameter type classes does

not seem very hard. Then it also make sense to address similar extensions, e.g., overlapping instances and functional dependencies, sooner rather than later.

Improving the engineering of specialized type rules will be addressed as the need arises. Considering how to extend further extensions, e.g., monad comprehrensions, is lowest on our list, but may, for example, be considered in a master thesis project, as is the development of heuristics for improving type error diagnosis, generally.

One issue that remains is "customer involvement". We welcome any contribution to our project of the following kind;

- examples of EDSLs that are in particular need of our treatment
- concrete feature requests from EDSL developers
- examples of programs beyond Haskell 98 that show that GHC type error diagnosis can be improved.

7 Conclusion

We have outlined a project that we plan to execute at the Software Technology group at Utrecht University to improve the state of the art in domain specific type error diagnosis, in particular for Haskell. We have shown what form this diagnosis takes when restricted to Haskell 98, and motivated an urgent need to go beyond that to modern-day Haskell. This paper is in a way also a call out to the field, to contribute type incorrect programs, and to point us to interesting, suitable EDSLs in Haskell for undergoing our treatment. In particular, we would like to come into contact with the developers of such EDSLs.

References

1. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.
2. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
3. A. Dijkstra and S.D. Swierstra et al. The Utrecht Haskell Compiler. `http://www.cs.uu.nl/wiki/UHC/WebHome`.
4. A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell Compiler. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.
5. N. el Boustani and J. Hage. Corrective hints for type incorrect Generic Java programs. In J. Gallagher and J. Voigtländer, editors, *Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*, pages 5–14. ACM Press, 2010.
6. N. el Boustani and J. Hage. Improving type error messages for generic java. *Higher-Order and Symbolic Computation*, 24(1):3–39, 2012. 10.1007/s10990-011-9070-3.
7. S. D. Swierstra et al. The Software Technology group at Utrecht University. `http://www.cs.uu.nl/groups/ST/twiki/bin/view/Center/`.

8. M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2011.
9. GHC Team. *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.
10. G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. In *Proceedings of the ACM SIGPLAN Haskell symposium, Tokyo, Japan*, pages 13–22. ACM, 2011.
11. K. Glynn, P. J. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.
12. J. Hage and M. Chakravarty. Private communication, January 2005.
13. J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.
14. J. Hage and B. Heeren. Strategies for solving constraints in type and effect systems. *Electronic Notes in Theoretical Computer Science*, 236:163 – 183, 2009. Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008).
15. J. Hage, B. Heeren, A. Middelkoop, et al. The Helium compiler. `http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome`.
16. J. Hage and M. Odersky. Private communication, September 2011.
17. B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. http://www.cs.uu.nl/people/bastiaan/phdthesis.
18. B. Heeren and J. Hage. Type class directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages*, pages 253 – 267, Berlin, 2005. Springer Verlag.
19. B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
20. F. Henglein et al. Hiperfit. Research project, `http://hiperfit.dk/`.
21. Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996.
22. J. Hughes, M. Sheeran, K. Claessen, and P. Jansson. Raw fp: Productivity and performance through resource aware functional programming. `http://wiki.portal.chalmers.se/cse/pmwiki.php/RAWFP/RAWFP`.
23. M. P. Jones. A theory of qualified types. In *ESOP'92: Symposium proceedings on 4th European symposium on programming*, pages 287–306, London, UK, 1992. Springer-Verlag.
24. S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative pluggable types for Java. *ACM Trans. Program. Lang. Syst.*, 32:4:1–4:37, February 2010.
25. J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber homepage, 2008. `http://www.timber-lang.org`.
26. M. Odersky. The Scala homepage, 2008. `http://www.scala-lang.org/`.
27. V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical Report HW-MACS-TR-0079, Herriot Watt University, Edinburgh, Scotland, Aug 2010.
28. D. Stewart and D. Coutts. Hackage, Cabal and the Haskell Platform. Presented at the Haskell Implementors Workshop 2010, Baltimore.
29. S. D. Swierstra, A. Rodriguez, A. Middelkoop, A. I. Baars, and A. Loeh et al. The Haskell Utrecht Tools (hut). `http://www.cs.uu.nl/wiki/HUT/WebHome`.

30. S.Doaitse Swierstra. Construct your own favorite programming language. Technical Report UU-CS-2009-029, Dept. of Inf. and Computing Sciences, Utrecht University, 2009.
31. W. Taha. Plenary talk iii domain-specific languages. In *Computer Engineering Systems, 2008. ICCES 2008. International Conference on*, pages xxiii –xxviii, nov. 2008.
32. D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *Journal of Functional Programming*, 21:333 – 412, 2011.
33. J. Weijers, J. Hage, and S. Holdermans. Security type error diagnosis for higher-order, polymorphic languages. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, PEPM '13, pages 3–12, New York, NY, USA, 2013. ACM.