Embedding and Parsing Combined for Efficient Language Design

Gergely Dévai, Dániel Leskó and Máté Tejfel
Faculty of Informatics, Eötvös Loránd University, Pázmány P. stny. 1/C, Budapest, Hungary

Keywords: Domain Specific Languages, Embedding, Concrete Syntax.

Abstract: One way to develop domain specific languages is to define their concrete syntax and create support for it using classical compiler technology (maybe with the support of language workbenches). A different way is to create an embedded language, which is implemented as a special library written in a host language. The first approach is usually too costly in the first phase of the language design when the language evolves and changes quickly. Embedded languages are more lightweight and support the language experiments better. On the other hand, they are not that convenient for the end-users as the standalone languages. This paper presents the lessons learnt from a DSL development research project in industry-university cooperation, that combined the advantages of the two approaches: the flexibility of embedding in the design phase and the convenience of a standalone language in the final product.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In special hardware or software domains the general purpose programming languages may not be expressive or efficient enough. This is why domain specific languages (DSLs) are getting more and more important. However, using classical compiler technology makes the development of new DSLs hard. The new language usually changes quickly and the amount of the language constructs increases rapidly in the early period of the project. Continuous adaptation of the parser, the type checker and the back-end of the compiler is not an easy job.

Language embedding is a technique that facilitates this development process. In this case a general purpose language is chosen, which is called the *host language*, and its parser and type checker are reused for the purposes of the DSL. In fact, an embedded language is a special kind of library written in the host language. The DSL programs in this setup are programs in the host language that extensively use this library. The library is implemented in such a way that its users *have the impression* that they are using a DSL, even if they are producing a valid host language program.

In this paper we use the so called *deep embedding* technique. Implementation of a deeply embedded language consists of

- *data types* to represent the AST,
- *front-end:* a set of functions and helper data types which provide an interface to build ASTs,
- *back-end:* interpreter or compiler that inputs the AST and executes the DSL program or generates target code.

Not all general purpose programming languages are equally suitable to be host languages. Flexible and minimalistic syntax, higher order functions, monads, expressive type system are useful features in this respect. For this reason Haskell and Scala are widely used as host languages. On the other hand, these are not mainstream languages. As our experience from a previous project (Dévai et al., 2010; Axelsson et al., 2010) shows, using a host language being unfamiliar to the majority of the programmers makes it harder to make the embedded DSL accepted in an industrial environment. In addition to this, error reporting and debugging are hard to solve in an embedded language.

For these reasons we have decided to create a standalone DSL as the final product of our current project. However, we did not want to go without the flexibility provided by embedding in the language design phase. This paper presents the experiment to combine the advantages of these two approaches.

This paper is based on a university research project initiated by Ericsson. The goal of the project is to develop a novel domain specific language that is specialized in the IP routing domain as well as the special hardware used by Ericsson for IP routing purposes.

This paper does not introduce the DSL created by

Dévai G., Leskó D. and Tejfel M..
Embedding and Parsing Combined for Efficient Language Design.
DOI: 10.5220/0004591802440250
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 244-250
ISBN: 978-989-8565-68-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

this project for two reasons. First, the language, being the result of an industry-university cooperation, is not publicly available at the moment. Second, the results presented in this paper concern the language development methodology used by the project. This methodology is general, and the concrete language it was applied to is irrelevant.

The most important lessons learnt from the experiment are the following. It was more effective to use an embedded version of the domain specific language for language experiments than defining concrete syntax first, because embedding provided us with flexibility so that we were able to concentrate on language design issues instead of technical problems. The way we used the host language features in early case studies was a good source of ideas for the standalone language design. Furthermore, it was possible to reuse the majority of the embedded language implementation in the final product, keeping the overhead of creating two front-ends low.

The paper is organized as follows. Section 2 introduces the architecture of the compiler. Then in section 3 we analyze the implementation activities using statistics from the version control system used. Section 4 presents related work, while section 5 concludes.

2 COMPILER ARCHITECTURE

The architecture of the software is depicted in figure 1. There are two main dataflows as possible compilation processes: *embedded compilation* (dashed) and *standalone compilation* (dotted).

The input of the embedded program compilation is a Haskell program loaded in the Haskell interpreter. What makes a Haskell program a DSL program is that it heavily uses the *language front-end* that is provided by the embedded DSL implementation. This front-end is a collection of helper data types and functions that, on one hand, define how the embedded program looks like (its "syntax"), and, on the other hand, builds up the *internal representation* of the program. The internal representation is in fact the *abstract syntax tree (AST)* of the program encoded as a Haskell data structure.

The same AST is built by the other, standalone compilation path. In this case the DSL program has it's own concrete syntax that is parsed. We will refer to the result of the parsing as *concrete syntax tree (CST)*. This is a direct representation of the program text and may be far from the internal representation. For this reason the transformation from the CST to an AST may not be completely trivial.

Once the AST is reached, the rest of the compilation process (optimizations and code generation) is identical in both the embedded and the standalone version. As we will see in section 3, this part of the compiler is much bigger both in size and complexity than the small arrow on figure 1 might suggest.

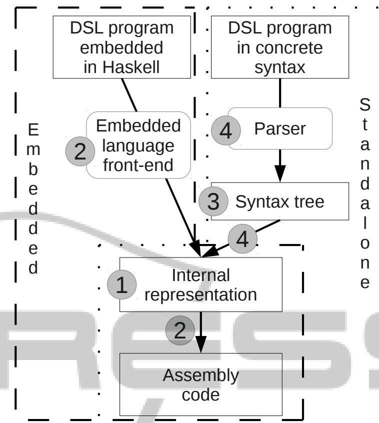

Figure 1: Compiler architecture.

The numbers on the figure show the basic steps of the workflow to create a compiler with this architecture. The first step is to define the data types of the internal representation. This is the most important part of the language design since these data types define the basic constructs of the DSL. Our experience has shown that it is easier to find the right DSL constructs by thinking of them in terms of the internal representation then experimenting with syntax proposals.

Once the internal representation (or at least a consistent early version of it) is available, it is possible to create embedded language front-end and code generation support in parallel. Implementation of the embedded language front-end is a relatively easy task if someone knows how to use the host language features for language embedding purposes. Since the final goal is to have a standalone language, it is not worth creating too fine grained embedded language syntax. The goal of the front-end is to enable easy-enough case study implementation to test the DSL functionality.

Contrarily, the back-end implementation is more complicated. If the internal representation is changed during DSL design, the cost of back-end adaptation may be high. Fortunately it is possible to break this transformation up into several transformation steps and start with the ones that are independent of the DSL's internal representation. In our case this part of the development started with the module that pretty prints assembly programs.

When the case studies implemented in the embed-

SCIENCE AND TECHNOLOGY PUBLICATIONS

ded language show that the DSL is mature enough, it is time to plan its concrete syntax. Earlier experiments with different front-end solutions provide valuable input to this design phase. When the structure of the concrete syntax is fixed, the data types representing the CST can be implemented. The final two steps, parser implementation and the transformation of the CST to AST can be done in parallel.

3 DETAILED ANALYSIS

According to the architecture in section 2 we have split the source code of the compiler as follows:

- *Representation:* The underlying data structures, basically the building data types of the AST.
- *Back-end:* Transforms the AST to target code. Mostly optimization and code generation.
- *Embedded front-end:* Functions of the embedded Haskell front-end which constructs the AST.
- *Standalone front-end:* Lexer and parser to build up the CST and the transformation from CST to AST.

The following figures are based on a dataset extracted from our version control repository[1]. The dataset contains information from 2012 late February to the end of the year.

Figure 2 compares the code sizes (based on the eLOC, effective lines of code metric) of the previously described four components. The overall size of the project was almost 9000 eLOC [2] when we summarized the results of the first year.

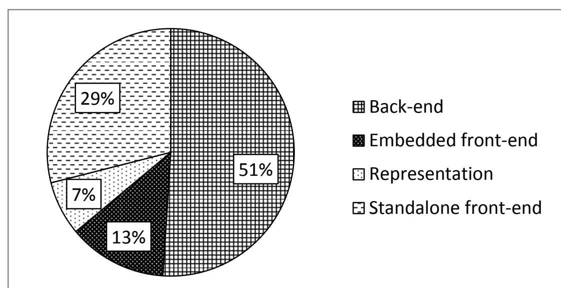

Figure 2: Code size comparison by components.

No big surprise there, the back-end is without a doubt the most heavyweight component of our language. The second place goes to the standalone front-end, partly due to the size of lexing and parsing codes[3]. The size of the embedded front-end is less than the half of the standalone's. The representation is the smallest component by the means of code size, which means that we successfully kept it simple.

Figure 3 shows the exact same dataset as figure 2 but it helps comparing the two front-ends with the reused common components (back-end, representation).

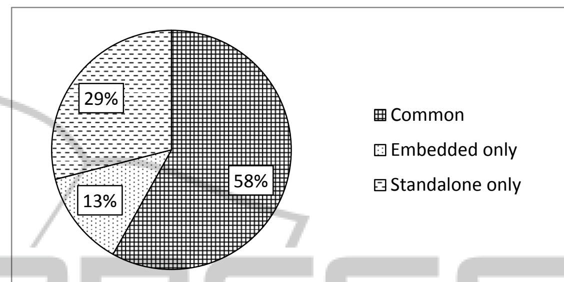

Figure 3: Code size comparison for embedded / standalone.

The pie chart shows that by developing an embedded language first, we could postpone the development of almost 30% of the complete project, while the so-called extra code (not released, kept internally) was only 13%.

Figure 4 presents how intense was the development pace of the four components. The dataset is based on the log of the version control system. Originally it contained approximately 1000 commits which were related to at least one of the four major components. Then we split the commits by files, which resulted almost 3000 data-points, that we categorized by the four components. This way each data-point means one change set committed to one file.

It may seem strange that we spent the first month of development with the back-end, without having any representation in place. This is because we first created a representation and pretty printer for the targeted assembly language.

The work with the representation started at late March and this was the most frequently changed component over the next two-three months. It was hard to find a proper, easy-to-use and sustainable representation, but after the first version was ready in early April, it was possible to start the development of the embedded front-end and the back-end.

The back-end and code generation parts were mostly developed during the summer, while the embedded front-end was slightly reworked in August and

[1] In this project we have been using *Subversion*.

[2] Note that this project was entirely implemented in Haskell, which allows much more concise code than the mainstream imperative, object oriented languages.

[3] We have been using the *Parsec* parser combinator library (Leijen and Meijer, 2001) of Haskell. Using context free grammars instead would have resulted in similar code size.

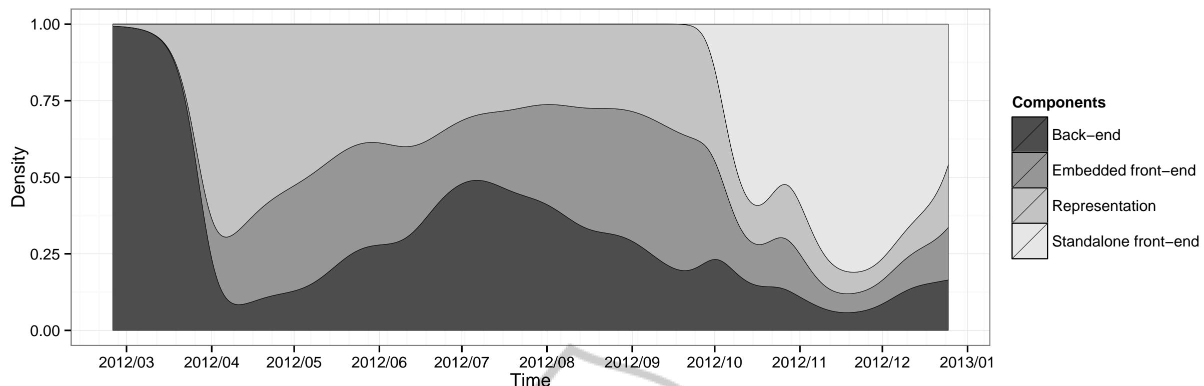

Figure 4: Development timeline.

September, because the first version was hard to use.

By October we almost finalized the core language constructs, so it was time to start to design the standalone front-end and concrete, textual syntax. This component was the most actively developed one till the end of the year. At the end of October we had a slight architecture modification which explains the small spike in the timeline. Approaching the year end we were preparing the project for its first release: Every component was actively checked, documented and cleaned.

4 RELATED WORK

Thomas Cleenewerck states that *"developing DSLs is hard and costly, therefore their development is only feasible for mature enough domains"* (Cleenewerck, 2003). Our experience shows that if proper language architecture and design methodology is in place, the development of a new (not mature) DSL is feasible in 12 months. The key factors for the success are to start low cost language feature experiments as soon as possible, then fix the core language constructs based on the results and finally expand the implementation to a full-fledged language and compiler.

Frag is a DSL development toolkit (Zdun, 2010), which is itself a DSL embedded into Java. The main goal of this toolkit is to support deferring architectural decisions (like embedded vs. external, semantics, relation to host language) in DSL software design. This lets the language designers to make real architectural decisions instead of ones motivated by technological constraints or presumptions. In our case there were no reason to postpone architectural decisions: It was decided early in the project to have an external DSL with a standalone compiler (see section 1). What we needed instead was to postpone their realization and keep the language implementation small and simple in the first few months to achieve fast and painless experiment/development cycles.

Another approach to decrease the cost of DSL design is published by Bierhoff, Liongosari and Swaminathan (Bierhoff et al., 2006). They advocate incremental DSL development, meaning that an initial DSL is constructed first based on a few case studies, which is later incrementally extended with features motivated by further case studies. This might be fruitful for relatively established domains. In our case the language design iterations were heavier then simple extensions. We believe that creating a full fledged first version of the language and then considerably rewriting it in the next iterations would have wasted more development effort than the methodology we applied.

At the beginning of our project a set of separate embedded language experiments were started, each of them dedicated to independent language features. These components were loosely coupled at that time, therefore gluing them to form the first working version was a relatively simple task to do. This kind of architecture is very similar to keyword based programming (Cleenewerck, 2003), where the complete DSL is formed by loosely coupled and independent language components. Later on our components became more and more tightly coupled due to the need of proper error handling and reporting, type and constraint checking.

Languages like Java, Ruby, MetaOCml, Template Haskell, C++, Scala are used or are tried to be used as implementation languages for developing new DSLs (Sloane, 2009; Freeman and Pryce, 2006; Cunningham, 2008; Czarnecki et al., 2003). These projects either used the embedded-only or the standalone-only approach and they all reported problems and shortcomings. We claim that many of these can be eliminated by combining the two approaches.

The Metaborg approach (Bravenboer and Visser,

2004; Bravenboer et al., 2005) (and many similar projects) extend the host language with DSL fragments using their own syntax. The applications are then developed using the mixed language and the DSL fragments are usually compiled to the host language. In our case the host language is only used for metaprogramming on top of the DSL, the embedding does not introduce concrete syntax and, finally, the host language environment is never used to execute the DSL programs.

David Wile has summarized several lessons learnt about DSL development (Wile, 2004). His messages are mostly about how to understand the domain and express that knowledge in a DSL. Our current paper adds complementary messages related to the language implementation methodology.

Based on Spinellis's design patterns for DSLs (Spinellis, 2001), we can categorize our project. The internally used embedded front-end is a realization of a piggyback design pattern, where the new DSL uses the capabilities of an existing language. While the final version of our language, which employs a standalone front-end, is a source-to-source transformation.

5 DISCUSSION AND CONCLUSIONS

5.1 Lessons Learnt

This section summarizes the lessons learnt from the detailed analysis presented in section 3.

Message 1: Do the language experiments using an embedded DSL then define concrete syntax and reuse the internal representation and back-end! Our project started in January 2012 and in December the same year we released the first version of the language and compiler for the industrial partner. Even if this first version was not a mature one, it was functional: the hash table lookups of the multicast protocol was successfully implemented in the language as a direct transliteration from legacy code. Since state of the art study and domain analysis took the first quarter of the year, we had only 9 months for design and implementation. We believe that using a less flexible solution in the language design phase would not have allowed us to achieve the mentioned results.

Message 2: Design the language constructs by creating their internal representation and think about the syntax later! The temptation to think about the new language in terms of concrete syntax is high. On the other hand, our experience is that it is easier to design the concepts in abstract notation. In our case this abstract notation was the algebraic data types of Haskell: The language concepts were represented by the data types of the abstract syntax tree. When the concepts and their semantics were clear there was still large room for syntax related discussions[4], however, then it was possible to concentrate on the true task of syntax (to have an easy to use and expressive notation) without mixing semantics related issues in the discussion. This is analogous to model driven development: It is easier to build the software architecture as a model and think about the details of efficient implementation later.

Message 3: Use the flexibility of embedding to be able to concentrate on language design issues instead of technical problems! Analysis of the compiler components in section 3 shows that the embedded front-end of the language is lightweight compared to the front-end for the standalone language. This means that embedding is better suited for the ever-changing nature of the language in the design phase. It supports the evolution of the language features by fast development cycles and quick feedback on the ideas.

Message 4: No need for a full-fledged embedded language! Creating a good quality embedded language is far from trivial. Using different services of the host language (like monads and do notation, operator precedence definition, overloading via type classes in case of Haskell) to customize the appearance of embedded language programs can easily be more complex then writing a context free grammar. Furthermore, advocates of embedded languages emphasize that part of the semantic analysis of the embedded language can be solved by the host language compiler. An example in case of Haskell is that the internal representation of the DSL can be typed so that mistyped DSL programs are automatically ruled out by the Haskell compiler. These are complex techniques, while this paper has stated so far that embedding is lightweight and flexible — is this a contradiction? The goal of the embedded language in our project was to facilitate the language design process: It was never published for the end-users. There was no need for a mature, nicely polished embedded language front-end. The only requirement was to have an easy-to-use front-end for experimentation — and this is easy to achieve. Similarly, there was no need to make the Haskell compiler type check the DSL programs: the standalone language implementation can-

[4] *"Wadler's Law: The emotional intensity of debate on a language feature increases as one moves down the following scale: Semantics, Syntax, Lexical syntax, Comments."* (Philiph Wadler in the Haskell mailing list, February 1992, see (Wadler, 1992).)

not reuse such a solution. Instead of this, type checking was implemented as a usual semantic analyzer function working on the internal representation. As a result of all this, the embedded forntend in our project in fact remained a light-weight component that was easy to adapt during the evolution of the language.

Message 5: Carefully examine the case studies implemented in the embedded language to identify the host language features that are useful for the DSL! These should be reimplemented in the standalone language. An important feature of embedding is that the host language can be used to generate and to generalize DSL programs. This is due to the meta language nature of the host language on top of the embedded one. Our case studies implemented in the embedded language contain template DSL program fragments (Haskell functions returning DSL programs) and the instances of these templates (the functions called with a given set of parameters). The parameter kinds (expressions, left values, types) used in the case studies gave us ideas how to design the template features of the standalone DSL. Another example is the scoping rules of variables. Sometimes the scoping rules provided by Haskell were suitable for the DSL but not always. Both cases provided us with valuable information for the design of the standalone DSL's scoping rules.

Message 6: Plan enough time for the concrete syntax support, which may be harder to implement than expected! This is the direct consequence of the previous item. The language features borrowed from the host language (eg. meta programming, scoping rules) have to be redesigned and reimplemented in the standalone language front-end. Technically this means that the concrete syntax tree is more feature rich than the internal representation. For this reason the correct implementation of the transformation from the CST to the AST takes time. Another issue is source location handling. Error messages have to point to the problems by exact locations in the source file. The infrastructure for this is not present in the embedded language.

5.2 Plans and Reality

Our original project plan had the following check points:

- By the end of March: State of the art study and language feature ideas.
- By the end of June: Ideas are evaluated by *separate* embedded language experiments in Haskell.
- By the end of August: The language with concrete syntax is defined.
- By the end of November: Prototype compiler is ready.
- December was planned as buffer period.

While executing it, there were three important diverges from this plan that we recommend for consideration.

First, the individual experiments to evaluate different language feature ideas were quickly converging to a joint embedded language. Project members working on different tasks started to add the feature they were experimenting with modularly to the existing code base instead of creating separate case studies.

Second, the definition of the language was delayed by three months. This happened partly because it was decided to finish the spontaneously emerged embedded language including the back-end, and partly because a major revision and extension to the language became necessary to make it usable in practice. As a result, the language concepts were more or less fixed (and implemented in the embedded language) by September. Then started the design of the concrete syntax which was fixed in October. At first glance this seems to be an unmanageable delay. However, as we have pointed out in this paper, it was then possible to reuse a considerable part of the embedded language implementation for the standalone compiler.

Third, we were hoping that, after defining the concrete syntax, it will be enough to write the parser which will trivially fit into the existing compiler as an alternative to the embedded language front-end. The parser implementation was, in fact, straightforward. On the other hand, it became clear that it cannot directly produce the internal representation of the embedded language. Recall what section 5.1 tells about the template features and scoping rules to understand why did the transformation from the parsing result to the internal representation take more time than expected. Therefore the buffer time in the plan was completely consumed to make the whole infrastructure work.

In brief, we used much more time than planned to design the language, but the compiler architecture of section 2 yet made it possible to finish the project on time.

5.3 Future

At the moment it is unclear what will happen to this compiler architecture in the future when more language features will be added.

Conclusions of this paper suggest that we continue with the successful strategy and experiment with new

language features by modifying, extending the embedded language and, once the extensions are proved to be useful and are stable enough, add them to the standalone language.

On the other hand, this comes at a cost: The consistency of the embedded and standalone language front-ends have to be maintained. Whenever slight changes are done in the internal representation, the embedded language front-end has to be adapted. We still do not know if this cost overwhelms the advantage that the embedded language offers for the language design.

Furthermore, since the standalone syntax is more convenient than the embedded language front-end, it might not be appealing to experiment with new language concepts in the embedded language. It also takes more effort to keep in mind two different variants of the same language.

Even if it turns out that it is not worth maintaining the embedded language front-end and it gets removed from the compiler one day, its important positive role in the design of the first language version is indisputable.

6 SUMMARY AND ACKNOWLEDGEMENTS

This paper evaluates a language development methodology that starts the design and implementation with an embedded language, then defines concrete syntax and implements support for it. The main advantage of the method is the flexibility provided by the embedded language combined by the advantages of a standalone language. We have demonstrated that most of the embedded language implementation can be reused for the standalone compiler.

We would like to thank the support of Ericsson Hungary and the grant EITKIC 12-1-2012-0001 that is supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund.

REFERENCES

Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A., and et al. (2010). Feldspar: A domain specific language for digital signal processing algorithms. In *IN: Proc. 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign. IEEE.*

Bierhoff, K., Liongosari, E. S., and Swaminathan, K. S. (2006). Incremental development of a domain-specific language that supports multiple application styles. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 67–78.

Bravenboer, M., Groot, R. D., and Visser, E. (2005). Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *In Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE05*. Springer Verlag.

Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. *SIGPLAN Not.*, 39(10):365–383.

Cleenewerck, T. (2003). Component-based dsl development. In *In Proceedings of GPCE03 Conference, Lecture Notes in Computer Science 2830*, pages 245–264. Springer-Verlag.

Cunningham, H. C. (2008). A little language for surveys: constructing an internal dsl in ruby. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 282–287, New York, NY, USA. ACM.

Czarnecki, K., O'Donnell, J. T., Striegnitz, J., and Taha, W. (2003). Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72.

Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelsson, E., Sheeran, M., Vajda, A., Lyckegård, B., and Persson, A. (2010). Efficient code generation from the high-level domain-specific language Feldspar for DSPs. In *ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems*.

Freeman, S. and Pryce, N. (2006). Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN Conference. OOPSLA'06*, pages 855–865, Portland, Oregon, USA.

Leijen, D. and Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world. *Electronic Notes in Theoretical Computer Science*, 41(1).

Sloane, A. M. (2009). Experiences with Domain-Specific Language Embedding in Scala.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99.

Wadler, P. (1992). Wadler's "Law" on language design. Haskell mailing list, http://code.haskell.org/˜dons/haskell-1990-2000/msg00737.html.

Wile, D. (2004). Lessons learned from real dsl experiments. *Sci. Comput. Program.*, 51(3):265–290.

Zdun, U. (2010). A dsl toolkit for deferring architectural decisions in dsl-based software design. *Information and Software Technology*, Vol.52(No. 7):733–748.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS