Language Design and Implementation via the Combination of Embedding and Parsing

Gergely Dévai(✉), Dániel Leskó, and Máté Tejfel

Faculty of Informatics, Eötvös Loránd University,
Pázmány P. stny. 1/C, Budapest, Hungary
deva@elte.hu

Abstract. Language embedding is a method to implement a new language within the framework of an existing programming language. This method is known to speed up the development process compared to standalone languages using classical compiler technology. On the other hand, embedded languages may not be that convenient for the end-users as standalone ones with own concrete syntax. This paper describes a method that uses the flexibility of language embedding in the experimental phase of the language design process, then, once the language features are mature enough, adds concrete syntax and turns the language to a standalone one. Lessons learnt from a project, run in industry-university cooperation and using the presented method, are discussed. Based on these results, a cost model is established that can be used to estimate the potential benefits of this method in case of future language design projects.

Keywords: Domain specific languages · Embedding · Parsing · Concrete syntax

1 Introduction

1.1 Motivation

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: It is time consuming and error prone.

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

J. Cordeiro and M. van Sinderen (Eds.): ICSOFT 2013, CCIS 457, pp. 131–147, 2014.
DOI: 10.1007/978-3-662-44920-2_9

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 [1,8] 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.

1.2 Project Background

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 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.

1.3 Main Messages

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 Sect. 3 we analyze the implementation activities using statistics from the version control system used. Section 4 presents related work, then Sect. 5 presents the main messages of the paper and a cost model to estimate benefits of the approach for future projects.

2 Compiler Architecture

The architecture of the software is depicted in Fig. 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 embedded language front-end module may contain complex functions to bridge the gap between an easy-to-use embedded language syntax and an internal representation suitable for optimizations and code generation. However, it is important that this front-end does not run the DSL program: It only creates its AST.

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

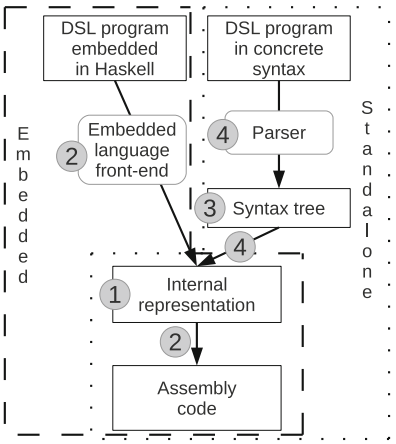

Fig. 1. Compiler architecture.

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 Sect. 3, this part of the compiler is much bigger both in size and complexity than the small arrow on Fig. 1 might suggest.

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 embedded 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 Sect. 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.

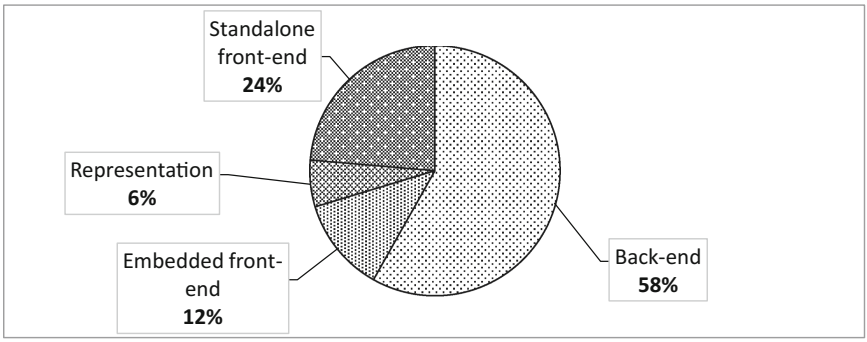

Fig. 2. Code size comparison by components.

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 LOC, lines of code metric) of the previously described four components. The overall size of the project was around 13 000 LOC[2] when we summarized the results of the first year.

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 about 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 Fig. 2 but it helps comparing the two front-ends with the reused common components (back-end, representation).

The pie chart shows that by developing an embedded language first, we could postpone the development of almost one fourth of the complete project, while the so-called extra code (not released, kept internally) was only 12 %. Note that these figures are based on the code size at the end of the project. The actual amount of work will be discussed later in Sect. 5.4.

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 file-change. All of these changes were weighted by the number of inserted lines. So at the end we got 75 000 data-points, that we categorized by the four components. This way each data-point represents one line insertion.

[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 [10] of Haskell. Using context free grammars instead would have resulted in similar code size.

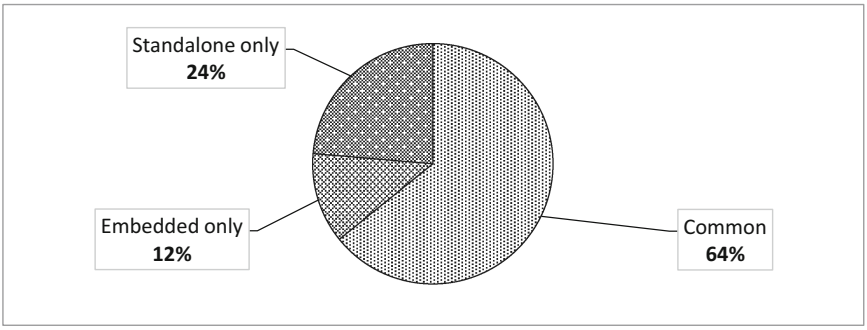

Fig. 3. Code size comparison for embedded / standalone.

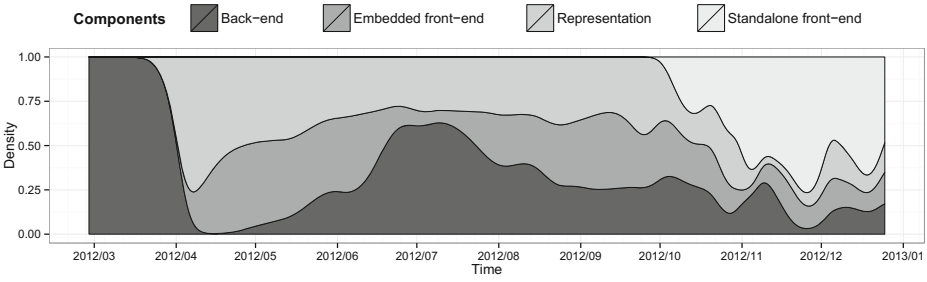

Fig. 4. Development timeline.

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 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. Early November we had a slight architecture modification which explains the small spike in the representation and back-end related parts. 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"* [5]. 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 [15], 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 Sect. 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 [2]. 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 [5], 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 [6,7,9,11]. 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 [3,4] (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 [14]. 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 [12], 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 Sect. 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.

[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 [13].)

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 Sect. 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 cannot 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 front-end 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 (e.g. 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 the development of a symbol table, which should store exact source positions in order to support good quality error messages, traceability, debugging and profiling. The symbol table is also useful for detecting the violations of scoping rules. This infrastructure is usually not (completely) present in an embedded language. To tell the truth, our embedded language implementation was not well prepared for the addition of this infrastructure. The lesson we have learnt here is that the embedded language implementation should be created keeping in mind that it will be turned to a standalone one later.

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 Sect. 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 Sect. 2 yet made it possible to finish the project on time.

5.3 Cost Model

The messages in Sect. 5.1 suggest that the presented method pays off if the flexibility of an embedded language provides more benefit in the language design phase than the additional cost of creating an embedded language front-end. This happens if the language experiments modify the code base intensively *and* the size of the embedded front-end is small enough compared to the size of the standalone front-end.

This section digs deeper into the analysis of this trade-off by setting up a cost model to predict the effort needed to create a new language by the following two methods:

- *Standard Method.* A standalone language is implemented from the beginning of the project, the parser is maintained in each iteration of the language design.
- *Combined Method.* The one described in this paper. That is, an embedded language is created for experimentation and the standalone language front-end is added when the language is fixed.

The first observation we have to make is that choosing the standard or the combined method does not influence the effort needed for the back-end implementation. This is because the back-end inputs the internal representation of the program, and the complexity to turn it to target code is not affected by the way this internal representation was built up: either embedding or parsing. As a consequence, the cost model has to deal only with the representation, embedded front-end and standalone front-end.

The effort needed to develop these components depends on their final *size* and their *variability*. We will use variability to measure the difficulty of reaching the final version of a given component: If the solution is straightforward, then the final version will likely be created by gradually adding new functionality to the code base. On the other hand, a development process that involves many experiments and dead-ends will have considerable amount of deletion and modification of existing code.

If we stick to the *insert* and *delete* operations widely used in software version control, we can treat modifications as the deletion of the old and insertion of the new versions. If $\#ins$ and $\#del$ denotes the number of insertions and deletions done to a component[5], then the following equations can be given for *size* and *variability*:

[5] The unit of measure can be anything from files to characters. Our statistics use lines.

$$size = \#ins - \#del$$

$$variability = \frac{\#del}{\#ins}$$

Zero variability means no deletions at all, while variability converging to 1 means that the amount of code added and later deleted overwhelms the size of the final product.

We argue that the *effort* needed to create a component is well characterized by the number of insertions during the development process, that is $\#ins$. From the two equations above, we get the following one:

$$effort = \frac{size}{1 - variability}$$

The effort is given by the size in case of projects with zero variability, while the effort converges to infinity, if variability gets close to 1.

Experiments in the language design phase introduce new language concepts, remove less successful ones or alter them. These changes alter the internal representation in the first place, but the front-end module (either embedded or standalone) has to be adapted to be able to build the code base and evaluate the experiment. How heavily does a representation change affect a front-end module? The amount might vary depending on the exact case, however, we argue that it will be proportional to the change in the representation: a small change in the representation induces a small change in the front-end and adding a completely new set of language concepts require an entirely new front-end module.

Let $variability_{exp}$ denote the variability experienced during the experimental phase. Later, when the language features and their semantics are fixed, one can expect considerably lower variability, since that part of the development process is well-specified. Ideally, the variability in that phase would be zero. However, even well specified and straightforward projects show a slight code variability due to refactoring steps and bugfixes. We will denote this value by $variability_{norm}$.

We are now prepared to calculate the effort needed for the whole project in case of standalone and combined strategies. (The component names *rep*, *sf* and *ef* will denote the representation, the standalone front-end and the embedded front-end respectively.)

$$effort_{standard} = \frac{size_{rep} + size_{sf}}{1 - variability_{exp}}$$

$$effort_{combined} = \frac{size_{rep} + size_{ef}}{1 - variability_{exp}} + \frac{size_{sf}}{1 - variability_{norm}}$$

The equation for the standalone case is the direct consequence of the equation for *effort* above. In the combined case, the variability of experimentation only affects the representation and the embedded front-end. The standalone front-end is free from the effects of the language experiments, since its development only starts when the language is fixed. For this reason the normal variability is applicable for that component.

The combined development method is beneficial if

$$effort_{combined} < effort_{standard},$$

which is equivalent to the following condition:

$$\frac{size_{ef}}{1 - variability_{exp}} + \frac{size_{sf}}{1 - variability_{norm}} < \frac{size_{sf}}{1 - variability_{exp}}$$

The first observation is, that the size of the representation component disappeared from the condition as the result of simplification, so the actual size of the language to be designed is irrelevant when choosing between the two design strategies.

Further transforming the condition results in the following form:

$$\frac{size_{ef}}{size_{sf}} < \frac{variability_{exp} - variability_{norm}}{1 - variability_{norm}}$$

Note that, if $variability_{norm}$ is close to its ideal zero value, then the right hand size is close to $variability_{exp}$. This suggests the following rule of thumb: *The combined method can be beneficial if the ratio of the sizes of the embedded front-end and the standalone front-end is smaller than the variability during the experiments.*

How to estimate this ratio and the two variabilities when starting a new language development project? The ratio of the front-end sizes depends on the host language to be used for the embedding and also on the tools, libraries to be used for lexing and parsing. It seems to be a good idea to implement both the embedded and standalone version of a toy language using the selected implementation language and toolset to estimate the required size ratio.

The value of $variability_{norm}$ mainly depends on the developer team and its ways of working. Measuring the variability in an earlier, relatively well-specified project done by the same team gives a good basis to estimate this value.

Estimation of the variability during the experimental phase is more difficult, but it is still possible if one considers how stable is the concept of the language to be created. If the specification is clear and the solution is straightforward, one can expect that $variability_{exp}$ will be close to $variability_{norm}$. On the other hand, a lot of room for experimenting with different language features will certainly lead to much higher variability. Rephrasing the question might help: *Will we delete half of the code that we create during the experiments in order to get to the version that is more or less fixed?* If the answer is yes, one can expect $variability_{exp}$ around 0.5.

Once there is an estimation for the three parameters, the condition discussed above can be used to make a decision how the language should be developed.

5.4 Was It Worth It?

As discussed in Sect. 5.1, our impression was that using the combined development method was actually beneficial for the project we have been working on. Does the cost model introduced above confirm this?

Analysing the statistics of the SVN repository used, we got that the final size of the embedded front-end was 1592 lines, while the standalone front-end consists of 3092 lines. The size ratio is therefore 0.515.

Regarding variability, 0.765 was measured for the representation and 0.725 for the embedded front-end. This confirms the assumption of the model that the variability of the representation is close to that of the front-end used in the experimental phase. The weighted variability of these two components is 0.738. In contrast, the variability of the standalone front-end was only 0.260.

The condition suggested by the cost model is therefore

$$\frac{1592}{3092} < \frac{0.738 - 0.260}{1 - 0.260},$$

which boils down to $0.515 < 0.646$. That is, also the cost model suggests that combining embedding and parsing was actually worth it in our case.

We have seen the ratio of each component in terms of the final code size earlier, on Fig. 2. Figure 5, in contrast, shows their relation in terms of the *effort* needed to develop them. Note that the standalone front-end shrank to less then its half, while the other three components grew equally. This is the consequence of the low variability of the standalone front-end, due to its postponed development.

One can argue that the benefit of the combined development strategy is negligible compared to the maintenance cost of the back-end: Every change in the representation induces a change in the back-end, which is by far the most heavyweight component.

On the other hand, a change in the representation only affects a limited segment of the back-end. To measure this impact precisely, we analysed the commits of our version control system. If a change to a back-end file was made in such a commit which also changed a representation file, then we consider this back-end change a result of the change in the representation, otherwise it is independent.

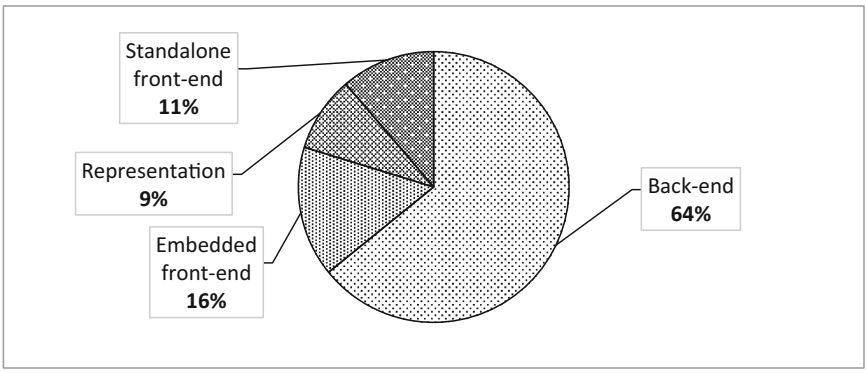

Fig. 5. Implementation effort by components.

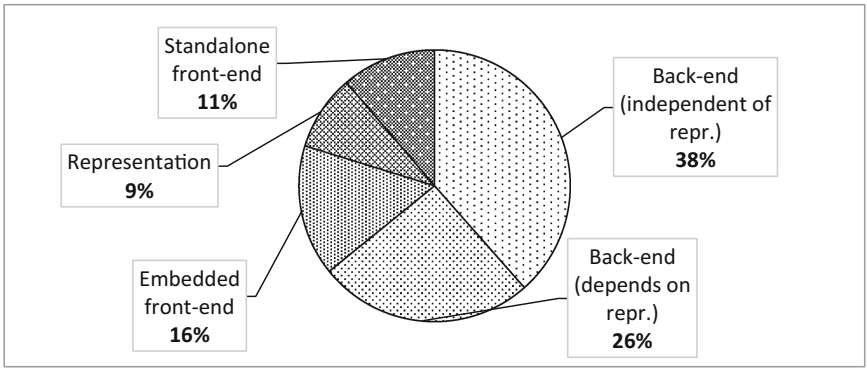

Fig. 6. Implementation effort by components (back-end partitioned).

The results show that only 40 % of the back-end changes are directly related to a change in the representation, the other 60 % is low level code generation and optimization related change. This is demonstrated by Fig. 6 where the back-end component (precisely its *effort*) is divided to a representation dependent and to a non-dependent part.

5.5 Future

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

In 2013 only minor new features have been implemented, not requiring intensive language experiments. These low variability implementation tasks are cheaper to be done directly in the standalone version of the language.

However, conclusions of this paper suggest that if we will need to develop more complex new features in the future we shall 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. Many trivial new features added only to the standalone front-end, like the ones we have developed in 2013, make the two front-ends diverge. We still do not know if the adaptation costs overwhelm 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

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. A cost model has been presented that tells if our method is rewarding in case of future projects.

Acknowledgements. 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

1. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A., et al.: Feldspar: a domain specific language for digital signal processing algorithms. In: Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign. IEEE (2010)
2. Bierhoff, K., Liongosari, E.S., Swaminathan, K.S.: Incremental development of a domain-specific language that supports multiple application styles. In: OOPSLA 6th Workshop on Domain Specific Modeling, pp. 67–78 (2006)
3. Bravenboer, M., de Groot, R., Visser, E.: MetaBorg in action: examples of domain-specific language embedding and assimilation using Stratego/XT. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 297–311. Springer, Heidelberg (2006)
4. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. SIGPLAN Not. **39**(10), 365–383 (2004). http://doi.acm.org/10.1145/1035292.1029007
5. Cleenewerck, T.: Component-based DSL development. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 245–264. Springer, Heidelberg (2003)
6. Cunningham, H.C.: 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, pp. 282–287. ACM, New York (2008). http://doi.acm.org/10.1145/1593105.1593181
7. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 51–72. Springer, Heidelberg (2004)

8. Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelsson, E., Sheeran, M., Vajda, A., Lyckegård, B., Persson, A.: 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 (2010)
9. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Companion to the 21st ACM SIGPLAN Conference. OOPSLA'06, pp. 855–865, Portland, Oregon, USA (2006). http://www.mockobjects.com/files/evolving_an_edsl.ooplsa2006.pdf
10. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Electron. Notes Theoret. Comput. Sci. **41**(1) (2001). Technical Report UU-CS-2001-35
11. Sloane, A.M.: Experiences with domain-specific language embedding in Scala. In: Lawall, J., Réveillére, L. (eds.) International Workshop on Domain-Specific Program Development (DSDP), Nashville, Tennessee, USA. vol. 7 (2008)
12. Spinellis, D.: Notable design patterns for domain-specific languages. J. Syst. Softw. **56**(1), 91–99 (2001). http://dx.doi.org/10.1016/S0164-1212(00)00089-3
13. Wadler, P.: Wadler's "Law" on language design. Haskell mailing list (1992). http://code.haskell.org/~dons/haskell-1990-2000/msg00737.html
14. Wile, D.: Lessons learned from real DSL experiments. Sci. Comput. Program. **51**(3), 265–290 (2004). http://dx.doi.org/10.1016/j.scico.2003.12.006
15. Zdun, U.: A DSL toolkit for deferring architectural decisions in DSL-based software design. Inf. Softw. Technol. **52**(7), 733–748 (2010). http://eprints.cs.univie.ac.at/2288/