The EDSL's struggle for their sources[*]

Gergely Dévai, Dániel Leskó, Máte Tejfel
Eötvös Loránd University, Faculty of Informatics

July 8-20, 2013

Abstract

Embedded Domain Specific Languages make language design and implementation easier, because lexical and syntactical analysis and part of the semantic checks can be completed by the compiler of the host language.

On the other hand, by the nature of embedding, EDSL compilers have to work with a syntax tree that stores no information about the source file processed and the location of the program entities within the source file. This makes it hard to produce user-friendly error messages and connect the generated target code with the source code for debugging and profiling purposes.

This lecture note presents this problem in detail and shows possible solutions. Section 2.2 also serves as a step-by-step guide for the lab session at the DSL2013 summer school.

1 Introduction

As software systems become more and more complex, using appropriate languages that provide the right abstraction level and domain-specific optimization possibilities is crucial to keep the time-to-market short, the maintenance costs low and the product performance high.

These observations lead to the application of domain specific languages in many different application areas. On the other hand, building applications using DSLs adds new challenges: Designing new languages and creating well performing compilers is hard, integrating many different languages and tools into a project may be difficult and DSLs usually lack the rich tool support (debuggers, profilers, static analisers) that widely used general purpose programming languages have.

This paper addresses some of these challenges. In particular, we concentrate on embedded domain specific languages (EDSLs), that are implemented as special libraries in a general purpose programming languages (called the host languages). In this setup, language design is simplified to a great extent compared

[*]Supported by EITKIC 12-1-2012-0001.

to standalone language development. On the other hand, good quality error reporting, possibility of source level debugging and profiling is much harder.

This latter deficiency of EDSLs is due to the fact that the compilers of these languages have no access to the source code of the program (unless the host languages have special support for this). This paper presents three possible solutions for the problem. These were developed in different EDSL projects that the authors of this paper were involved in lately. One of these projects is Feldspar [2, 7], which stands for Functional Embedded Language for Digital Signal Processing and Parallelism. It was originally initiated by Ericsson AB and run by Chalmers Univerity of Technology in Gothenburg and ELTE University in Budapest. The other project, called Miller [8], was initiated by Ericsson Hungary and is run at ELTE University. The objective of this project is to create a domain specific language for architectures with complex programmable memory hierarchies. The topic of the third project [6] is an embedded language to express formal specifications of programs and correctness proofs. All the three projects created embedded languages using Haskell as the host language.

The rest of this section introduces the concept of embedding and gives the details of the source code accessing problem. Section 2 presents a solution using preprocessing with standard tools, while section 3 describes a more advanced possibility with syntax tree manipulation. Finally, section 4 shows how to combine the development of an embedded language with its standalone version.

1.1 EDSLs

DSLs are usually categorized as *external* or *internal*. External DSLs are implemented as a stand alone language with own syntax and compiler, without any particular connection to any existing language. On the other hand, internal DSLs are created within the framework of another (usually general purpose) programming language, which is called the host language. The relation between an internal DSL and its host language can be of many sort. A detailed overview can be found in [11].

In this paper we consider a specific kind of internal DSL implementation strategy that Hudak [9] named as *domain specific embedded language (DSEL)* and is also called as *embedded domain specific language (EDSL)*.

An EDSL is a library written in the host language. EDSL programs are therefore programs in the host language that intensively use that library. The border between traditional libraries and EDSLs is not always clear, but it is an important feature of EDSLs that they have some kind of domain semantics in addition to their meaning as plain host language programs.

There are two types of EDSL: *shallow* and *deep* embeddings. In case of a shallow embedding, running the EDSL program as a host language program computes the result of the EDSL program. On the other hand, executing a program of a deeply embedded language as a host language program only creates the abstract syntax tree of the EDSL program. This AST is then usually further processed by the interpreter or compiler of the EDSL to execute the program or to generate target code. In the rest of the paper we will only focus on deeply embedded DSLs. Creating a deeply embedded DSL consists of the following steps:

- Definition of the data types of the abstract syntax tree. We will also refer to these data types as *internal representation*.
- Implementation of a front-end: a set of helper data types and functions that can be used to build up the abstract syntax tree. The purpose of this front-end is to provide a user-friendly way of writing EDSL programs. This frontend determines how EDSL programs will "look like", therefore one might say that it defines the EDSLs "syntax".
- Implementation of a back-end that processes the syntax tree: a code generator to transform the EDSL program to target code or an interpreter to execute it.

Compared to a standalone language, an EDSL is usually easier to develop:

- Since the EDSL has no own syntax, there is no need for lexer and parser: These tasks are done by the host language compiler.
- If the host language has expressive enough type system, it is also possible to encode much of the semantic rules of the EDSL in the types of the abstract syntax tree elements and frontend functions. This way the semantic analysis is partly done by the host language compiler too.
- The full power of the host language can be used to write meta programs on top of the EDSL. As EDSL programs are valid host language programs, EDSL program fragments can be freely combined and parametrized.

These advantages make embedding particularly suitable for language design experiments. More on this aspect will be presented in section 4.

These observations are more-or-less true also for the comparison of EDSLs with other internal language implementation techniques, like Metaborg [4]. In case of Metaborg-style embeddings, one defines stand alone syntax for the DSL, but the DSL code fragments are written in host language source files. These mixed-language source files are then processed by the compiler of the DSL and the DSL fragments are translated to pure host language code. In the next step the compiler of the host language is used to create an executable.

Haskell is particularly well-suited to be a host language: Its syntax is minimal and is flexible enough to support different EDSL syntax styles. The type system of the language is advanced, allowing the language designer to encode many EDSL semantic rules in the types.

1.2 Accessing source code

Compilers of traditional, standalone languages have full access to the source files. Lexing and parsing keep track of the locations and string values of the tokens and the syntax tree can be annotated with this information. This annotation is then used for several different purposes:

- Quality error messages. The error messages contain exact (file, row, column) locations of the error. It may also name the entities (functions, variables etc.) that are involved in the error, using the same names that appear in the source file.

- Readable target code. Many DSL compilers translate the source code to another textual programming language instead of machine code. Programmers usually want to read this generated code and understand its connections to the original DSL code. To help this understanding process, it is helpful if the generated code uses the same names for variables, functions as the source. It may also be a good idea to add comments to the generated code showing connected DSL code fragments.

- Finding the connection between the source and target code is important not only for humans but also for software: In order to a show the active source code instruction during a debugging session or to show the values of variables it is necessary to provide the debugger with a mapping between the source and target code.

- The above mentioned mapping is also necessary to project profiling results back to the source level. This enables profilers to show performance bottlenecks in the source code or to provide runtime statistics on function or instruction level.

While any parsing based DSL development methods (standalone languages, Metaborg-style internal languages) have all the necessary information for the above tasks, EDSLs usually lack this information. The reason for this is simple: The EDSL compiler's input is the abstract syntax tree that was created by running the EDSL program as a host language program. As this syntax tree is not the result of parsing, location and textual information is not present. In order to have that in an EDSL syntax tree, the host language should provide constructs to ask for location and text of any program fragment.

Summarizing sections 1.1 and 1.2, one is faced with the following trade-off: On the positive side, EDSLs can use the host language compiler to solve lexical, syntactical and (partly) semantic analysis. Furthermore, the host language becomes a powerful meta programming layer on top of the EDSL. On the negative side, EDSL compilers completely lack source location and text information which prevents creating good quality error messages and connecting the generated target code for human understanding, debugging and profiling.

2 Preprocessing

2.1 Concept

Preprocessing is the process of scanning and modifying source code before the compiler inputs it. The most widely used preprocessor is the CPP (C preprocessor) which is used with many languages besides C and C++, including Haskell.

It supports macro definitions (`#define`), conditional compilation (`#if`, `#ifdef` etc.), inclusion of other files into the source file (`#include`), which is also used instead of a proper modul system in C/C++.

There is a clone of the C preprocessor tailored a bit to the Haskell language: *cpphs* [1]. It accepts quotes and backticks in macro names to match the Haskell identifier lexical rules. However, operator symbols still cannot be used in macro names. Also, this tool is easier to integrate into compiler projects using Haskell as the implementation language than the traditional C preprocessor.

When compiling (or interpreting) Haskell sources using preprocessor directives, additional parameters are needed, for example:

```
ghci -cpp MyFile.hs
```

This will call the traditional C preprocessor, while the following uses cpphs:

```
ghci -cpp  -pgmPcpphs  -optP--cpp MyFile.hs
```

Preprocessing can also be used as a lightweight solution to the source code access problem of EDSLs. The transformation steps of an EDSL implementation can be summarized as follows:

- Extend the data types of the abstract syntax tree to be able to store source file names and line numbers, symbol names in the source etc.
- Add more parameters to selected interface functions to be able to pass these pieces of information.
- Create macros that generate these additional values automatically and publish these macros to the user instead of the original interface functions.

2.2 Example

This section presents examples on using the C preprocessor to reflect symbol names in the EDSL code in the generated target code and to make helpful error messages. It is also a the step-by-step guide for the lab session for this topic at the summer school.

2.2.1 An example EDSL

The language used as an example, called *Simple* contains the integer and boolean types, variables, basic arithmetic and logic operations and assignment. An example program:

```
correct :: Simple
correct = do
    x <- int
    x .= 10
    x .= x + 20 - 2 * x
    y <- bool
    y .= true && not y || false
```

You can try it during the lab session as follows:

```
ghci Examples.hs
*Main> compile correct
int var0;
bool var1;
var0 = 10;
var0 = ((var0+20)-(2*var0));
var1 = ((true&&(!var1))||false);
```

The generated code contains generated variable names, which makes it harder to read and is really annoying in the error messages:

```
*Main> compile wrong1
Errors:
Non lvalue found on the left hand side of an assignment: (var0-1)
```

The task of the lab session is to fix these problems, but first let us overview the implementation of the EDSL before modifying it.

2.2.2 The implementation of Simple

The implementation consists of three files in the *Simple* dirctory:

- *Representation.hs*: The data types of the abstract syntax tree of the language.
- *Frontend.hs*: The types and functions that can be used in *Simple* programs.
- *Compiler.hs*: Functions to check programs for errors and to generate target code.

A *Program* consists of *Declaration*s and *Instruction*s. A declaration contains a variable of some type *a*, where *a* is a *Supported* type of the language. An instruction is an assignment, consisting of two *Expression*s: the left and right hand sides of the assignment. Expressions are either *Literal*s, *Var*iables or compound expressions built up by arithmetic or logic operators.

Note that expressions are typed, this means that type errors in *Simple* programs will be reported already by the Haskell compiler:

```
wrong3 :: Simple
wrong3 = do
    x <- int
    x .= true
```

The error message for this program is:

```
Examples.hs:31:10:
    Couldn’t match expected type ‘Prelude.Int’
                with actual type ‘Prelude.Bool’
    Expected type: Simple.Representation.Expression Prelude.Int
      Actual type: Simple.Representation.Expression Prelude.Bool
    In the second argument of ‘(.=)’, namely ‘true’
    In a stmt of a ’do’ block: x .= true
```

The frontend of the language instantiates the *Num* class for *Expression Int* to provide integer literals and basic arithmetic in the language. The *true* and *false* functions are the boolean literals and the standard `(&&)`, `(||)` and `not` operations of the Haskell *Prelude* are redefined as the boolean operations of *Simple*.

The instructions in a *Simple* program are written in a monadic environment. The monad is called *Simple* and is a state monad with a state that collects the declarations and instructions of the program, and an *Integer* used to generate unique names for the declared variables.

The *int* and *bool* are monadic functions resulting in *Expression Int*s and *Expression Bool*s, so that they can be used to declare variables in the DSL programs. These functions *get* the actual state of the program, create a new *Declaration* with a *Variable* of the desired type inside, add this declaration to the program, increment the integer used as unique identifier in variable names and finally *put* the modified state back into the monad.

The *(.=)* operator can be used in the language to write an assignment operation. This function is also monadic, it adds the new assignment instruction to the state.

The *compile* function runs the state monad in order to obtain the abstract syntax tree of the program and calls the *compile'* function defined in the *Compiler* module to generate code.

As defined in *Compiler.hs*, the *Result* of the compilation is a list of *String*s, which is either *Code* or *Errors*.

Compilation is monadic, uses a state monad with the *Result* type as the state. The *addInstruction* and *addError* functions help adding new target code lines or error messages to the state. If an error occurs, the code lines generated so far and to be generated later are omitted and only the error messages are collected.

The *compileDeclaration*, *compileInstruction* and *compileExpression* monadic functions are used to generate code for declarations, instructions and expressions respectively. The *compileInstruction* function also reports an error when anything but a variable is found on the left hand side of an assignment.

2.2.3 Elimination of the generated variable names

The first goal is to modify the language frontend so that programmers can set variable names that will appear in the generated code:

```
correct :: Simple
correct = do
    x <- int "x"
    x .= 10
    x .= x + 20 - 2 * x
    y <- bool "y"
    y .= true && not y || false
```

Add a parameter of type *String* to the frontend functions *int* and *bool* and use this name instead of the generated one. The result of the compilation should now look like:

```
int x;
bool y;
x = 10;
x = ((x+20)-(2*x));
y = ((true&&(!y))||false);
```

On the other hand, this solution is inconvenient for the programmers and it is also easy to mess things up if the Haskell names and DSL names of variables diverge: `x <- int "y"`.

For this reason, create a header file called `simple.h` and move the import directives at the beginning of `Examples.hs` into it. Include the header in the example file: `#include "simple"`. From now on, you have to pass the `-cpp` option to *ghci* so that it calls the C preprocessor before parsing.

Now, define the `int` and `bool` macros in the header file, each with one parameter. The macro call `int(x)` has to expand to `x <- int "x"`. Rewrite the examples so that they use the newly defined macros instead of the `int` and `bool` frontend functions:

```
correct :: Simple
correct = do
    int(x)
    x .= 10
    x .= x + 20 - 2 * x
    bool(y)
    y .= true && not y || false
```

This way the error messages reporting invalid assignments become a little bit more helpful, because they refer to the variables by their original names in the source code.

2.2.4 Adding file names and line numbers to error messages

First, write the function

```
checkDeclarations :: [Declaration] -> ResultM ()
```

in `Compiler.hs` to find duplicate variable names in the declaration list. Use the `addError` function to report error. Call this function in the first line of the `compile''` function:

```
checkDeclarations (declarations prg)
```

Now the `wrong2` example should also result in an error message:

```
Variable x is redefined.
```

This error message could be more helpful if indicated the source file and the lines that caused the error:

```
Examples.hs, line 22: Variable x is redefined. Earlier definition
is in line 20.
```

In order to achieve this you need to implement the following modifications:

- Add two new parameters of types `String` and `Int` to the functions `int` and `bool` to be able to pass the file name and the line number of the variable definition.
- Use the `__FILE__` and `__LINE__` macros in the definition of the `int` and `bool` macros to pass these new parameters.
- In `Representation.hs`, add two new constructor parameters to the `Variable` constructor of types `String` and `Int`. Adapt the compiler's code to this change.
- In the `int` and `bool` functions pass the two new parameters to the `Variable` constructor in order to store the file and line information in the abstract syntax tree.
- In the `checkDeclarations` function use the new constructor parameters of `Variable` to extend the error message with useful information.

2.2.5 Better error messages about assignments

The techniques seen in the previous section can be applied to make the error message about incorrect assignment instructions more user friendly. In order to do this, we need to turn the `(.=)` operator to a macro. This, unfortunately, will make the EDSL syntax less pretty:

```
correct :: Simple
correct = do
    int(x)
    let(x, 10)
    let(x, x + 20 - 2 * x)
    bool(y)
    let(y, true && not y || false)
```

On the other hand, we can make the assignment related error message look like this:

```
Examples.hs, line 14: Non lvalue found on the left hand side of
an assignment: (x-1)
```

In order to achieve this, you have to add new parameters to the `(.=)` function, implement the `let` macro in the header file, add new constructor parameters to `Assign` and use them in the error message inside the `compileInstruction` function.

2.2.6 Further possibilities

The same technique can be used for example to simplify the definition of *Simple* programs. Instead of writing

```
correct :: Simple
correct = do
  ...
```

one might prefer using this syntax:

```
simple(correct)
  ...
```

This way we can further enrich the error messages with information about the function in which the error is located.

2.3 Evaluation

To summarize the techniques we have seen in the previous sections, we conclude that the advantage of this solution is its simplicity and also that it only requires easy-to-use and standard tools like the C preprocessor.

On the other hand, all the well-known pitfalls of the textual replacement of macro expansion make this solution dangerous. Another disadvantage is that eventual Haskell error messages will refer to the code after macro expansion, while the user edits the one with macros.

The approach is also limited, and it affects the syntax of the EDSL as we have seen in the examples so far.

3 Syntax tree manipulation

In case of languages with own concrete syntax and a parser, it is easy to create a mapping between the source code and the target code, because the compiler gets the source file and analyses it from character to character, so it gets the position for each syntactical unit instantly and can store it in the syntax tree. But, as described in section 1.2, this is not the case for embedded languages, they use the compiler of the host language to produce it's own embedded representation, the embedded compiler will not get any information about the source code.

This section presents a solution to this problem, which is more heavy weight than simple preprocessing, but is also more powerful. The idea is to perform

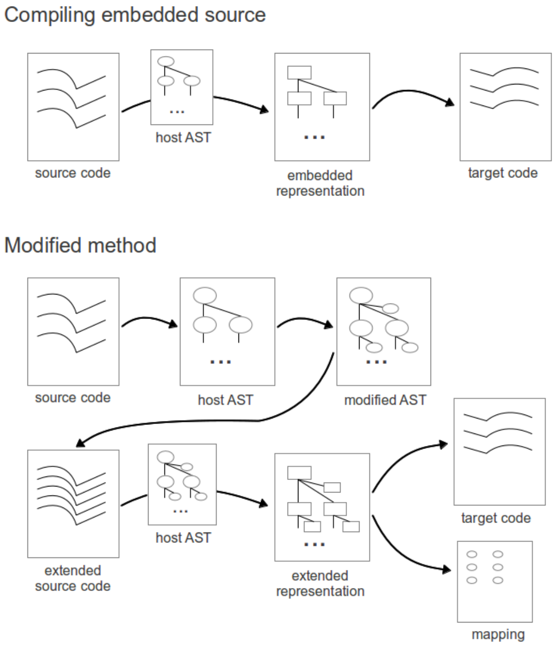

Figure 1: Compiling embedded source

DSL-specific preprocessing, using the compiler of the host language. This way we gain access to the position of each syntactical units, and can store it in the embedded syntax tree. For this, the we need to extend the internal representation (abstract syntax tree) and the frontend library. Using the extra location information, the compiler can create a mapping between the stored positions and the corresponding position of the target code.

3.1 Extended compilation

Compiling an embedded source is done by the following process: The interpreter of the host language analyses the source code, the program is executed as a host language program and builds the internal representation of the DSL program. Than the EDSL compiler generates the target code from the internal representation.

In order to add location information to the internal representation, the compilation workflow becomes more complex. First of all we need the host syntax tree of the embedded source (the host AST). After obtaining that, a transformation needs to get the positions of each syntactical units and extend the host AST with further nodes that represent wrapper functions. If we transform the modified syntax tree back to embedded source, every necessary source position will appear. In this solution the interface library, provided for the programmers and the embedded representation need to be extended with the wrapper functions and the corresponding data types.

During the code generation we need to save each position from the node of the embedded syntax tree and match it up with the corresponding position of the target code to complete the mapping. Figure 1 illustrates the differences of the original and the extended compilation process.

3.2 Transformation

The first step of the described solution is the manipulation of the host AST. During this whenever we find a node that represents a syntactical unit we need to extend the host AST with further nodes representing a label function. The label function holds the source position of the syntactical unit that is being labelled. If we didn't violate the syntactical rules of the host language, the extended host AST can be easily transformed back to source code where every syntactical unit's position will appear as an argument of the corresponding label function. The transformation itself is independent of the embedded language, it depends only on the host language. So if we define the transformation for a host language, it can be used for every embedded language which uses that host language.

3.3 Code generation

Code generation also becomes a bit more complex, because it has to to calculate the absolute position of parts of the generated target code and produce a mapping between the target and the embedded source. For this purpose, we need additional information to be able to generate code from each node of the embedded syntax tree: *the measure of the indentation*, *the absolute row and column position*, where the code generation should start from, *the absolute row and column positions*, where the code generation ends. This information, being spread among the nodes of the abstract syntax tree is categorized as follows:

- downward spread: information that every child node gets with the same value (*eg. measure of indentation*)

- upward spread: an information that the parents get from their child (*eg. generated target code*)

- state-like: an information that the node get from its parent and use it to calculate other information (*eg. absolute start row and column positions*)

The code generator uses wrapping nodes and other nodes in the abstract syntax tree differently: Nodes that represent language constructs are turned into target code, while wrapper nodes are used to produce the location mapping between the source and target files.

3.4 Embedding the *While* language

The *While* language is a very simple imperative language which consists a sequence of simple statements such as assignment and control statements (if-then-else and while loop). Programmers can use logical constants and expressions eg. true, false, comparison, negation and basic arithmetic operations like addition, subtraction and multiplication. We choose this language, because it is not too complex, so we can focus on the mapping problem.

First of all, we need to define a data type, that describes the abstract representation of *While* language programs. A possible implementation of the embedded syntax is the following.

```
data Program where
  (:=)       :: Variable a -> Expr a -> Program
  Declare    :: [Variable a] -> Program
  Sequence   :: [Program] -> Program
  Loop       :: Expr Bool -> Program -> Program
  IfThenElse :: Expr Bool -> Program -> Program -> Program
  Skip       :: Program

Data Expr a where
  Plus    :: Expr Int -> Expr Int -> Expr Int
  And     :: Expr Bool -> Expr Bool -> Expr Bool
  Compare :: Expr Int -> Expr Int -> Expr Bool
  Value :: (Show a) => a -> Expr a
  Var     :: Variable a -> Expr a

data Variable a where
  Variable :: Name -> Variable a

type Name = String
```

The *Program* data type was defined as a generalised algebraic data type. This way it is possible to use type variables in the constructor parameters that do not appear as type variables of the *Program* data type itself.

So far, we defined the internal representation of the While language, but using directly the defined data constructors is not convenient. So we need to define an interface for the programmers, that hides the representation of the language and helps them to build the syntax tree conveniently.

```
class Compare a where
  (<) :: a -> a -> Expr Bool

class Equal a where
  (==) :: a -> a -> Expr Bool

class Logical a where
  (&&), (||) :: a -> a -> Expr Bool
  (!) :: a -> Expr Bool
```

We make *Expr Bool* an instance of these type classes so that programmers can write logical expressions in a convenient way. There is an opened question: Why did not we use the *Eq* type class that is provided by Haskell's Prelude module? The answer is simple, the type signature does not fit:

```
class Eq a where
  (==) :: a -> a -> Bool
```

In this case, if we want to examine if two integers or boolean values are equal, the result will be a *Bool*, but we need an *Expr Bool* instead. However, when it comes to arithmetic operations, we can use the *Num a* typeclass. If we make the *Expr Int* type an instance of the *Num a* type class we can even use integer literals in the arithmetic expressions of the DSL.

```
instance Num (Expr Int) where
  (+) = Plus
  (-) = Minus
  (*) = Mul
  fromInteger i = Value $ fromInteger i
```

In general, before the code generation process, the EDSL's compiler is allowed to make transformations on the embedded syntax tree to optimize it. This is not done in our case to make the example simple. Therefore the compilation phase contains only code generation.

```
class CodeGenerator a where
  generate :: a -> Int -> String
```

The *CodeGenerator* type class is used to generate target code from simple nodes. The *generate* function takes a node from the syntax tree and an indentation value and produces the corresponding target code. We use the *StateMonad*, to store the generated target code. The *code* function takes a *String* as an argument and puts it to the state of the monad. We give some example instances of the *CodeGenerator* typeclass:

```
instance CodeGenerator Program
 where
  generate ((:=) (Variable name) expr) indent = cSource
   where
    ((), cSource) = flip runState "" $ do
     code $ indenter indent ++ name
     code $ " = " ++ generate expr 0 ++ "\n;"
...
instance CodeGenerator (Expr a)
 where
  generate (Plus lhs rhs) indent = cSource
   where
    ((), cSource) = flip runState "" $ do
```

```
      code $ "(" ++ generate lhs 0 ++
      code $ ") + (" ++ generate rhs 0 ++ ")"
...
```

3.5 Extending the language

So far we have presented a possible way to embed the *While* language into Haskell. However using the illustrated method the compiler does not have any information about the source code. In this part we extend our language, step-by-step, as mentioned in section 3.1.

First, we extend the internal representation of the language and the frontend with wrapper nodes and functions. In the next step we apply a transformation, that manipulates the Haskell syntax tree in order to inject the positions of the syntactical units into the DSL program source code. Because we have extended the programmers' interface and the embedded representation, and during the transformation we did not break any syntactical rule of Haskell, the source code that is pretty printed from the transformed host syntax tree will result in valid code either in Haskell as well as in the (extended) *While* language.

New data constructors, so called wrapper nodes, is defined in the abstract syntax tree to store the positions of the embedded source. These constructors take a source position, and a node to be wrapped as an argument. In our case we need only three wrapper nodes:

```
type SrcLoc = ((Int, Int), (Int, Int))
type Name   = String

data Program where
...
  LabProg :: SrcLoc -> Program -> Program

data Expr a where
...
  LabExpr :: SrcLoc -> Expr a -> Expr a

data Variable a where
  Variable :: Name -> Variable a
  LabVar   :: SrcLoc -> Variable a -> Variable a
```

SrcLoc represents the wrapped node's position in the source. In the first tuple the start row and column positions are stored, while the second tuple stores the end row and column positions.

The frontend needs to be extended with functions that represent the wrapper nodes. For this purpose we created the *Label a* typeclass that consists the *label* function which takes a source position and a node from the embedded syntax tree. It wraps the node with the corresponding wrapper data constructor. Below is the definition of the *Label* typeclass and some of the necessary instances:

```
class Label a where
  label :: ((Int, Int), (Int, Int)) -> a -> a

instance Label Program where
  label = LabProg

instance Label (Expr a) where
  label = LabExpr
```

Using this modification the representation will be capable of storing information about the source code, but the question remains: how can we label the syntactical units with their source position? A tool is needed that can syntactically analyse Haskell source and build a syntax tree containing the necessary information. For this purpose we have chosen the *haskell-src-exts* package. After syntactically analysing the source, it can produce the host AST. All we need to do is to identify the syntactical units and extend them with new nodes that represents the previously defined *label* function.

First of all, we need a helper function that retrieves source information from a node of the Haskell syntax tree. For this purpose the *Location a* type class is defined:

```
class Location a where
  getStartLine :: a SrcSpanInfo -> Int
  getStartCol  :: a SrcSpanInfo -> Int
  getEndLine   :: a SrcSpanInfo -> Int
  getEndCol    :: a SrcSpanInfo -> Int
```

We instantiates this type class with the types that build the host syntax tree.

Every node contains its position in the source code. The information is stored in a value with a type of *SrcSpanInfo*. The *haskell-src-exts* package provides helper functions to retrieve this information. Another function needs to be defined, to extend the syntax tree with further nodes, that represents the previously introduced *label* function. The new function has two arguments, the second will be the node that is to be wrapped, the first argument will be the source position of the wrapped node. The next example is simplified to make it easier to read:

```
wrap :: Exp SrcSpanInfo -> Exp SrcSpanInfo
wrap exp =
  (App
    (App
      (Var
        (UnQual (Ident "label"))
      )
      (Tuple
        [ Tuple [ startLinePosition
                , startColumnPosition ]
        , Tuple [ endLinePosition
                , endColumnPosition ]
        ]
      )
    )
    exp
  )
```

So far we have defined the transformation on a single node. We need to apply this on every node in the host AST. The *Transform a* type class is responsible for this. From the point of view of the transformation, the significant nodes are the nodes having type *Exp*, especially the function applications. The instance of the transformation function for nodes of other types is the identical mapping.

```
class Transform a where
   transform :: (a SrcSpanInfo) -> (a SrcSpanInfo)

instance Transform Exp
...
  transform x@(App _ _ _) = wrap $ transformRec x

transformRec (App inf fun arg) =
        App inf (transformRec fun) (transform x)
transformRec x = transform x
```

Note that the *transformRec* function can handle any function application with arbitrary number of arguments.

So far we managed to store each syntactical unit's position in the embedded representation, however we do not know the nodes' position in the generated target code. So our task is to calculate each node's position during the code generation. For this purpose we need additional information for each node:

- an absolute location, where the code generation starts (state information),
- an absolute location, where the code generation ends (upwards propagated information),
- and the measure of the indentation (downwards propagated information).

Upwards propagated information is stored in a record data structure. The *Result* record has three fields

- source: contains the generated target code so far,
- mapping: mapping generated so far
- position: the target's code absolute position generated from the latest node from the embedded syntax tree.

Now we can define the *DebugInfo* and *DebugInfo1* type classes which describe the modified code generation. The latter one is used when the current node contains a list.

```
type Location = (Int, Int)

class DebugInfo a where
  generateDebugInfo :: a -> Int -> Location -> Result

class (DebugInfo a) => DebugInfo1 a where
  generateDebugInfo' :: [a] -> Int -> Location -> Result
```

The original version of the code generator uses a *State monad*, where the state is the generated code, and the result value is unit. In the extended version, the *State monad* is used again, but the state will be a tuple with three members:

- the generated source so far,
- the absolute starting line position,
- the absolute starting column position.

We need to access these members during the whole process of code generation, so the entire procedure needs to be monadic.

Another monadic function is needed, that calculates the ending position of the code generated from the actual node of the embedded representation. With these helper functions the *DebugInfo a* typeclass can easily be implemented for the data types defined in the internal representation. However, as we pointed out earlier, wrapper nodes and language constructs are handled differently. In the case of a wrapper node our task is to call the monadic wrapper function with the wrapped node and lift the result into the monad. After that we need to retrieve the target code's position from the *Result* record's *position* field, pair it with the corresponding position in the source, then extend the list in the *mapping* field with this value. On the other hand, if we are dealing with a node that represents a language construct, the *code* function is used recursively for code generation on each of the children nodes.

At the end of the compilation, the Result record will contain the generated target code, and the mapping between the target code and the embedded source.

3.6 Summary

Debugging existing source code is not a simple task, even in case of general purpose languages. In case of domain specific languages, source level debugging is more complicated by the increased abstraction level. The generated target code can be debugged and the results have to be mapped back to the DSL level. This task is even harder in case of embedded programming languages, the mapping between the generated target code and the source code is missing. This section presented a general method to extend an existing embedded language and its compiler to be able to produce this mapping.

The extension consists of the following elements:

- a transformation, that manipulates the host syntax tree,
- extended version of the internal representation and frontend containing wrapper nodes and functions,
- modified code generation to keep track of the location of the target code for each node in the abstract syntax tree.

The extended compilation workflow is able to produce the necessary mapping. The method can be used for every embedded language given that we have a tool to easily produce and maipulate the syntax tree of the host language.

4 Embedding and parsing combined

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.

As described in section 1.1, *language embedding* is a technique that facilitates this development process. 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 [5, 3] 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.

Because of this, it is reasonable to create a standalone DSL as the final product of DSL projects. However, it would be beneficial to make use of the flexibility provided by embedding in the language design phase. This section of the paper presents our experience from an experiment to combine the advantages of these two approaches. The findings are 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.

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.

This section is organized as follows. Section 4.1 introduces the architecture of the compiler. Then in section 4.2 we analyse the implementation activities using statistics from the version control system used. Section 4.3 summarizes the learnt lessons.

4.1 Compiler Architecture

The architecture of the software is depicted in figure 2. 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 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 4.2, this part of the compiler is much bigger both in size and complexity than the small arrow on figure 2 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.

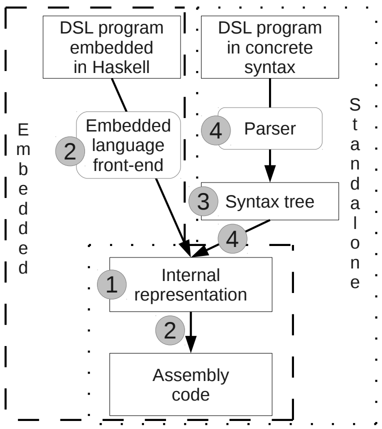
DSL program embedded in Haskell
DSL program in concrete syntax
Embedded
Standalone
2 Embedded language front-end
4 Parser
3 Syntax tree
4
1 Internal representation
2
Assembly code

Figure 2: Compiler architecture.

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.

4.2 Detailed Analysis

According to the architecture in section 4.1 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 3 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.

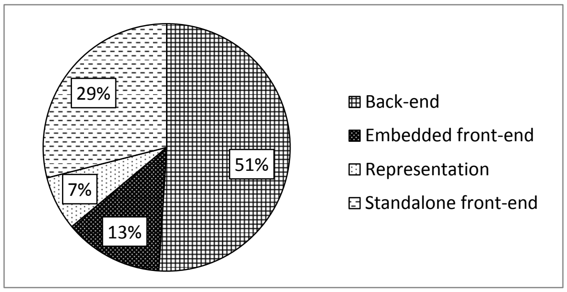

Figure 3: Code size comparison by components.

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 4 shows the exact same dataset as figure 3 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 30% of the complete project, while the so-called extra code (not released, kept internally) was only 13%.

Figure 5 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.

[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.

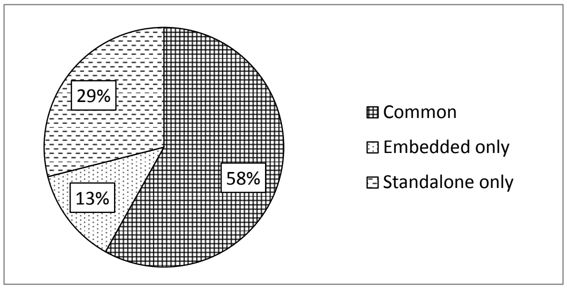

Figure 4: Code size comparison for embedded / standalone.

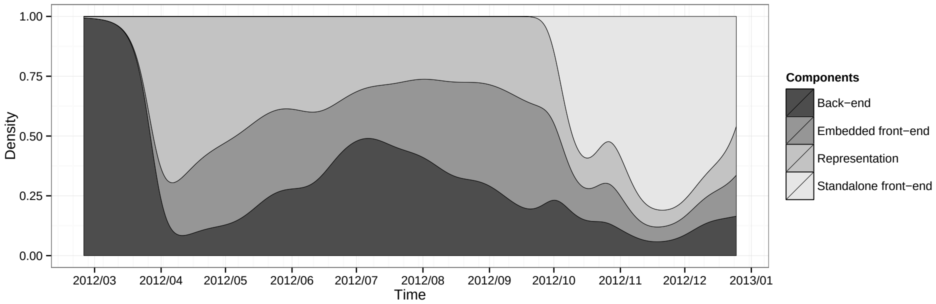

Figure 5: Development timeline.

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. 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.

4.3 Lessons Learnt

This section summarizes the lessons learnt from the detailed analysis presented in section 4.2.

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 4.2 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,

[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 [12].)

while we 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 analyser function working on the internal representation. As a result of all this, the embedded frontend 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.

4.4 Plans vs 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 previously, 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 4.3 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 4.1 yet made it possible to finish the project on time.

4.5 Future

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

Conclusions 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.

5 Summary

This paper deals with the problem that EDSLs' compilers have no access to their source code, which would be important for good quality error messages, debugging and profiling. Three different solutions are outlined.

Section 2 discussed how to use standard source code preprocessing tools like the C preprocessor to add the missing location information to the abstract syntax tree of the EDSL.

Next, in section 3, we have generalized the preprocessing solution: The method presented there extends the AST with and the language frontend with wrappers and reuses the host language compiler to inject the location information into the EDSL's AST. The code generator is then able to produce a mapping to connect the generated target code with the corresponding source code fragments.

Finally, section 4 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. Experience from a project using this methodology shows that most of the embedded language implementation can be reused for the standalone compiler.

References

[1] cpphs: Haskell implementation of the C preprocessor, `http://projects.haskell.org/cpphs/`.

[2] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and A Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[3] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Anders Persson, Mary Sheeran, Josef Svenningsson, András Vajda,

and et al. 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*, 2010.

[4] Martin Bravenboer, René De Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *Generative and Transformational Techniques in Software Engineering*, pages 297–311. Springer, 2006.

[5] G. Dévai, M. Tejfel, Z. Gera, G. Páli, G. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, and A. Persson. 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.

[6] Gergely Dévai. Embedding a proof system in haskell. In *Central European Functional Programming School*, pages 354–371. Springer, 2010.

[7] Gergely Dévai, Máté Tejfel, Zoltán Gera, Gábor Páli, Gyula Nagy, Zoltán Horváth, Emil Axelsson, Mary Sheeran, András Vajda, Bo Lyckegård, et al. 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.

[8] Gergely Dévai, Máté Tejfel, and Dániel Leskó. Embedding and parsing combined for efficient language design (accepted for publication at icsoft-ea). 2013.

[9] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

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

[11] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):31, 2008.

[12] Philip Wadler. Wadler's "Law" on language design. Haskell mailing list, http://code.haskell.org/~dons/haskell-1990-2000/msg00737.html, 1992.