Analysing embedded domain specific languages in Haskell from Core

A case study on Yard, a Parsec-like parser combinator library

P. T. Jager
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.t.jager@student.utwente.nl

ABSTRACT

Many programming languages are created for a specific purpose. These Domain Specific Languages are designed to solve one problem in a specific domain and solve it well. However, the implementation of these languages requires a lot of work. Therefore a lot of these languages are embedded into existing general purpose languages. Although this saves a lot of work implementing the language, it also has its drawbacks. One of these is that the compiler for the genral purpose language has no knowledge of the specific domain and therefore can not optimise the generated code using this knowledge. In this paper we explore the usage of Core-to-Core transformations through GHC plugins as a method of embedding domain specific knowledge in the Haskell compiler. We show that it is possible to analyse which code is domain specific and which is general Haskell code. We also provide suggestions as to which specific problems an EDSL author might encounter when embedding domain specific knowledge using Core-to-Core transformations and what present solutions to these problems from an EDSL author's point of view.

Keywords

Embedded Domain Specific Language, Domain Knowledge, Optimisation, Haskell, Core, GHC

1. INTRODUCTION

Many programming languages are created for a specific purpose. These *Domain Specific Languages* (DSLs) [5] are designed to solve problems from a specific domain and solve them well. Common examples of DSLs include HTML, SQL, LaTeX and the formula language in Microsoft Excel. These DSLs enable their users to easily describe problems in the domain-specific terms they are used to. However, the implementation of these DSLs requires the implementation of a complete stack of lexer, parser, pretty printer etc. to interpret or compile the DSL.

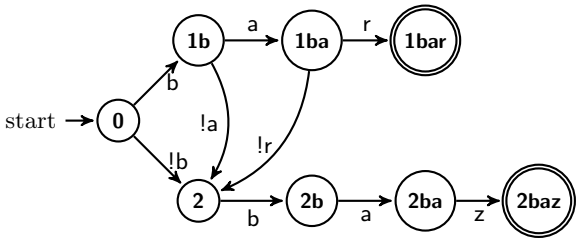
To overcome this, many DSLs are embedded into an existing general purpose programming language (GPL) such as Java or Haskell. These DSLs, we call *Embedded* DSLs (EDSL). Embedding the DSL has the advantage that now both the expressive power of the containing GPL and its compiler can be used. However, the compiler of the GPL does not have any knowledge of the domain described by the EDSL. It can therefore not do any domain-specific optimisations on programs described with the EDSL, beyond those which can be derived from the structure of the EDSL as described in the GPL. This leaves the compiled code less efficient than it could be.

Consider for example an EDSL — in the GPL Haskell — which describes parsers (for more on this EDSL see Section 2.4). It has primitives to construct parsers, such as `string :: String -> Parser String`, which either accepts a string that precisely matches its argument, or fails and functions to combine these parsers, such as choice: `<|>:: Parser a -> Parser a -> Parser a`, which tries the left parser and if that fails, returns the result of the right parser, which is either a parsed `a` or a fail.

We consider the following program, which parses either the string "bar" or the string "baz".

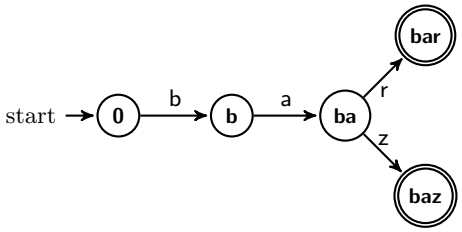
```
1  barOrBaz :: Parser String
2  barOrBaz = (string "bar") <|> (string "baz")
```

If we compile this code using the Haskell compiler and then try to parse the string "baz", the parser `string "bar"` is tried first. When this parser fails on the last character 'z' the characters already consumed ("ba") have to be restored and the parser `string "baz"` is tried. The state machine of the compiled Haskell program for `barOrBaz` is, overly simplified, the following. Note that we have omitted the failure states and the actions which restore the already consumed characters.



Since the compiler has no knowledge of the semantics of our domain, it does not know that the states *(1b, 2b), (1ba, 2ba)* could be merged and that state *2* could be omitted, resulting in the following state machine, which now does not require any backtracking when parsing.

21th Twente Student Conference on IT June 23^{d}, 2014, Enschede, The Netherlands.



When, instead of this small example, we consider a larger and more realistic grammar, it is clear that the amount of backtracking quickly becomes prohibitive, even though a properly generated parser does not need any backtracking.

In this research, we have done a case study on analysing an EDSL in Haskell. We provide a proof-of-concept compiler plugin (Section 2.2), which shows that it is possible to analyse a program described using an EDSL from within *Glasgow Haskell compiler* (GHC) [9]. The EDSL we analyse is a parser combinator library called Yard, more on this in Section 2.4.

This paper makes the following contributions:

- We show that it is possible to analyse the Core (see Section 2.1) generated by GHC to find the grammar described using Yard (Section 4).
- We show that it is possible to recognise the Haskell code which is interleaved with the grammar found (Section 4).
- An evaluation of optimising an EDSL from Core from an EDSL author's point of view (Section 8).

2. BACKGROUND

2.1 Core

GHC transforms Haskell source code into an intermediate language called Core [9, section 3.1] (Figure 1). Core is a very minimal language, which has the same expressive power as Haskell, but with very few syntatic constructs. In GHC, optimisations are implemented as a pipeline of Core-to-Core transformations. In Figure 1, we have grayed out the constructs in Core that are not relevant for this paper. To give some intuition of Core for the remainder of this paper the relevant ones are shortly explained:

Var is a variable, which has both a name and a type, like `Prelude.(+) :: Num a => a -> a -> a` or `foo :: Maybe String`

CoreBind is a binding of a left hand side (name) to a right hand side (expression). This can either be a non recursive binding, or a list of mutually recursive bindings, that is, bindings that reference each other, e.g. `ones = 1 : ones`.

CoreExpr

- **Var Var** is an expression of just a variable. Note that Var could be a function.
- **Lit Literal** is an expression of just a literal, e.g. `5` or `'c'`
- **App CoreExpr CoreExpr** is an application of a function to its arument, i.e. `f a`
- **Lam Var CoreExpr** is a lambda expression binding a name in a given expression, i.e. `\a -> f a`

2.2 GHC plugins

Since version 7.2.1 of the GHC Haskell Compiler [1] it is possible to define compiler plugins which are run as optimisations. These plugins can then do Core-to-Core transformations on the module that is currently being compiled. The plugin adds a function of type `ModGuts -> CoreM Modguts` (Figure 1) to the Core-to-Core optimisation pipeline. This function can then change the Core-binders in the ModGuts and thereby change the Core program in the module.

2.3 Parsers as functions

In functional languages, parsers can be modeled as functions which can then be combined using higher order combinator functions [6]. These combinator functions have parsers as arguments and result in a new parser which is a combination of the given arguments. A very short example of this approach is given in Section 1. These parser functions and their combinators are a *monad*, an algebraic structure. This allows us to treat parser functions as computations with context. For our parsers, this context is threefold: it holds the input to be consumed, the accumulated result and whether or not the parser has failed.

Failure needs to be tracked because when two parsers are combined, e.g. using `parser1 >> parser2`, where `>>` is sequence, this chain is a new parser. This parser only succesfully consumes input when `parser1`, followed by `parser2`, both succesfully parse input. It is therefore necessary that failure is propagated.

Using this monadic structure, it is also possible to have dependent parsers, that is, parsers that depend on parameters which are the result of a previous parser. Because of this dependent nature, it is possible to construct parsers which parse context sensitive grammars [8].

2.4 Yard

The EDSL this case study focusses on is Yard, that is Yet Another Research DSL. It is a DSL which offers monadic parser and combinator functions as described in Section 2.3. Yard is inspired by Parsec [8], which is a popular parser combinator library for Haskell. Unlike Parsec, in which parsers with lookahead larger than 1 have to be annotated, Yards parsers have infinite lookahead by default. In Parsec this decision was made for performance reasons. Yard however was created to be optimised and therefore does not require this annotation.

In Figure 2, an overview of the Yard EDSL is given using its functions' type signatures. It exists of a few basic parsing functions, some of which are parametric, and a few functions which combine parsers into new parsers.

3. PROBLEM STATEMENT

Describing context sensitive grammars using the monadic parser combinator approach from Section 2.3 is a very natural and powerful way to do so. However, implementing a complete language to express parsers this way is a very large and time consuming task. It would be better to reuse parts of an existing general purpose programming language and implement the parser functions as a library. Such a library can then be used to express parsers in the containing GPL and the complete range of functionality of the containing GPL can be used to expand the expressive power of the parser functions.

In the above case, the parser library is considered a DSL, embedded in a GPL. By embedding the DSL, it is both faster to implement and more powerful. However, by em-

```
type Var = -- ...
data ModGuts  = ModGuts { mg_binds :: [CoreBind], ... }
data CoreBind = NonRec Var CoreExpr | Rec [(Var, CoreExpr)]
data CoreExpr = Var Var | Lit Literal  | Type Type | App CoreExpr CoreExpr  | Lam Var CoreExpr
              | Let CoreBind CoreExpr  | Case CoreExpr Var Type [CoreAlt] | Cast CoreExpr Coercion
              | Coercion Coercion | Tick CoreTickisch CoreExpr
```

Figure 1. The Core language as seen by GHC plugins

```
-- parser functions
item   :: Parser Char          -- reads the next char
zero   :: Parser a             -- always fails
char   :: Char -> Parser Char  -- matches the given Char, or fails
alpha  :: Parser Char          -- matches any letter, or fails
string :: String -> Parser String -- matches the given string, or fails
digit  :: Parser Char          -- matches any digit, or fails
number :: Parser String        -- matches one or more digits, or fails
-- combinator functions
>>     :: Parser a -> Parser b -> Parser b -- chains two parsers
<|>    :: Parser a -> Parser a -> Parser a
    -- trys the left parser first and returns its result when it succeeds, or returns the result of the
     right parser when the left failed.
>>=    :: Parser a -> (a -> Parser b) -> Parser b -- monadic bind, runs the first parser and then uses
    its result as parameter for the second.
many   :: Parser a -> Parser [a] -- zero or more
optional :: a -> Parser a -> Parser a
    -- trys the parser and when its succeeds returns its result, or a default value when it fails
```

Figure 2. Overview of Yard

bedding the DSL, it is also compiled using the compiler of the containing GPL. This compiler has no knowledge of the domain of the DSL, which means it can not do any domain-specific optimisations, beyond that which can be derived from the structure of the EDSL.

This is true for any EDSL. Any compiler of a GPL can not extract more information of a domain beyond that which can be derived from the structure of the (implementation of an) EDSL. All compilers therefore compile to (potentially) less than optimal code when compiling code which uses an EDSL. Depending on the EDSL and GPL the performance impact of this could be significant, in either computational time, code size, or both. This problem is therefore not only relevant in Haskell or for a parser combinator EDSL, but for all EDSLs in general.

Even though the performance hit of EDSLs is potentially prohibitive, there have not been any real structural solutions, which aid EDSL designers in bringing their domain-specific knowledge into the compiler to the extend that the compiler can optimise their EDSL. We discuss some existing approaches in Section 5.

We aim to find a natural and easy way for EDSL authors to aid the GPL compiler in optimising their EDSL. Ultimately, this approach allows EDSL authors to bundle their domain-specific knowledge with their EDSL and have the compiler run domain specific optimisations as one of the compiler passes.

4. GHC PLUGINS FOR EDSL OPTIMISATIONS

In this paper, we propose a method to analyse shallow embedded DSLs from within the Haskell compiler using GHC plugins (see Section 2.2). This method could then be used to optimise the EDSL from within a GHC plugin. We show a proof-of-concept implementation (freely available [7]) of an analyser for the EDSL Yard (Section 2.4).

In our analysis we transform a program, descibed using Yard, to a grammar tree (Figure 4). This datatype describes grammars in terms we understand. Note how the first twelve constructors correspond to the twelve functions from Yard (Figure 2) and the types `Var` and `Literal` are those discussed in Section 2.1. Haskell constructs we do not understand, such as `let` or `case` are saved, packed in `GUnknown`. Below we expand on the grammar tree and how it is constructed from the Core AST, which has already been introduced in Section 2.1.

When analysing programs, described using Yard, several kinds of parser functions and interleaved Haskell constructs and functions need to be considered. In ascending order of complexity:

'simple' parsers Parsers that are described using only fully applied functions from Yard, e.g. `string "foo" >> (char 'c' <|> (string "bar" >> char 'd'))` .

Non-terminals Parsers described using multiple non-terminals. Such as the grammar described in Figure 3.

Lambda expressions Parsers which contain lambda constructs, i.e. `p1 >>= \x -> p2 <|> return x`. Understanding these requires knowledge of the semantics of the lambda Core construct.

Dependent parsers Parsers which depend on the production of a previous parser, that is, those which are

```
1 foo :: Parser String
2 foo = string "foo" >> string "bar"
3 baz :: Parser String
4 baz = foo >> string "baz"
```

Figure 3. Gammar with non-terminals

context sensitive, e.g. `alpha >>= \a -> char a`.

General functions Parsers which are interleaved with general functions, e.g. `\x -> string $ f x`, where `f` is any function.

We analyse programs by recursively walking the Core AST as shown in Figure 5. The function `unwind` unwinds an `App` to a `Stack` of arguments and the function to which they are applied on top. The function `readVar` reads a variable and resolves it to a Grammar, applying possible arguments from the argument list. Possible variables are the functions from Yard, which are mapped to one of the first twelve constructors from `Grammar`, non-terminals, or other functions.

Using this method we analyse all Core constructs and functions that are meaningfull in relation to Yard, i.e. functions from Yard itself, functions which result in type `Parser a` and lambda constructs. We observe that other general functions exist and we also analyse their variables if those are parsers. Other Core constructs are simply packed, placed in the grammar and ignored (see Section 7).

5. RELATED WORK

Farmer et al. have done considerable amounts of work on Core-to-Core transformations with their HERMIT tool [4]. This tool allows user interaction with Core directly within the GHC pipeline. Sculthorpe et al. have also done some evaluation of the HERMIT tool in which they found it works well for Core-to-Core transformations, but that it could benefit from more high level transformation functions [11]. HERMIT uses the KURE rewrite engine [12], a Haskell library which allows for typed data transformations, to do transformations on Core.

Adams et al. have used Core-to-Core optimisations in HERMIT to optimise the Scrap Your Boilerplate (SYB) EDSL [2]. Their results eliminate the performance impact of generic programming with SYB by removing run-time type inspection. They do so by constant propagation and dead code removal. Ultimately we aim to do optimisations which transform the optimised program, as discussed in Section 1.

In an (yet) unpublished paper Farmer and Gill have described and implemented a DSL to describe Core-to-Core transformations for HERMIT [3]. This allows for the optimisations to be described directly in the source files describing the EDSL, instead of seperate files for HERMIT.

GHC also features RULES pragmas, which are simple rewrite rules Haskell developers can embed in their source files to instruct GHC to rewrite certain function calls to other function calls [10]. Although rewite rules are a powerful tool, they are limited to local, small step transformations. The rewrites they describe are however expressed in a syntax that is quite similar to the expressions rewritten. This is an advantage from an EDSL authors point of view, as it eliminates the need to study Core.

6. CONCLUSIONS

We have shown that it is possible to analyse an EDSL from Core. We have shown that it is possible te recognise which Core constructs correspond to functions from the EDSL and which do not. We have also shown that it is possible to safely deal with those that do not and that, due to the recursive nature of our transformation functions, it is even possible to analyse their arguments.

7. FUTURE WORK

It would be interesting to use the method proposed in this paper to implement an EDSL optimisation GHC-Plugin, which actually implements an optimisation as a Core-to-Core transformation.

It would also be interesting to expand the understood Core constructs. Most notably the `Let` and `Case` constructs. Which would allow the analyses of parsers that use either let-bindings (e.g. `let x = y in string x`) or case statements.

As we discuss in Section 8, analysing the complete Core representation of a module is not a trivial task and might be a lot to ask from an EDSL author. It would therefore be benefitial to have a library which would aid EDSL authors in their optimisations, by tackling some of the problems discussed.

8. DISCUSSION

Analysing an EDSL from Core is not a trivial task. One of the main problems we have run into during the development of our proof-of-concept is that there does not appear to be an easy way to check if a variable is of a certain type or if it is in a certain module. It is possible to get the type or module of encountered variables, but there does not appear to be an easy way to check it against a type or module known by name. In our proof-of-concept we have overcome this by using the pretty printer to get string representations of the encountered variables and their types, however this is not a solution which is viable in a real world application. It would be benefitial for EDSL authors if they could simply check for a type or module, i.e.:

```
1 f :: CoreExpr -> a
2 f (Var v) | v `inModule` "Text.Yard" -> ...
3           | v `ofType` "Text.Yard.Parser a" ->
    ...
```

Another problem is that it appears to be difficult to construct `CoreExpr`s. For example, if we want an expression for the variable `foo :: Int -> Int` or the AST for `foo 2` there is no easy way to create these. The ability to do so would be beneficial as it would allow for easy replacement of `CoreExpr`s similar to GHC RULES (see Section 5). A library aimed at helping EDSL authors should provide this functionality.

Another issue is that due to the structure of the Core AST it is not easy to get all arguments for a function. Since the leftmost function in an expression is in the bottom leftmost leaf of the branch in the Core AST corresponding to that expression, it is necessary to traverse the AST upwards to rediscover all of its arguments. In Figure 6 we illustrate this for the expression `f 3 4 5`. EDSL authors might benefit from a library function which allows for easy inspection and changing of function arguments, which in themself might be other functions. In our proof-of-concept we have worked around this issue by transforming the entire Core AST into a stack.

```
data Grammar = GDontCare {-item-} | GFail {-zero-} | GChar String | GAlpha | GString String | GDigit
             | GNumber | GSequence Grammar Grammar | GChoice Grammar Grammar | GBind Grammar Grammar
             | GMany Grammar | GOptional String Grammar
             | GUnknown (CoreExpr) {- Packed unknown Core construct -}
             | GUnpack Var Grammar {- GHC function which unpack literals -}
             | GLiteral Literal {- A literal -}
             | GNonTerminal Var [Grammar] {- Local parser function -}
             | GLambda Literal Grammar  {- Lambda expression -}
             | GVariable Var [Grammar] {- Other function -}
```

Figure 4. Grammar

```
type Stack = [CoreExpr]
unwind :: CoreExpr -> Stack
readVar :: Var -> [Grammar] -> Grammar

compile :: Stack -> (Stack, Grammar)
compile (x:xs) = case x of
  Lit l   -> (xs, GLiteral l)
  Var v   -> case arity $ varType v of a
    | a > 0  -> let (xs', args) = (\(args, xs') -> (xs', map (compile . unwind) args)) $ splitAt a xs
                in (xs', readVar v args)
    | a == 0 -> (xs, readVar v [])
  Lam l e -> (xs, GLambda l $ compile $ unwind e)
  _       -> (xs, GUnknown x)
```

Figure 5. Compiling Core to Grammar

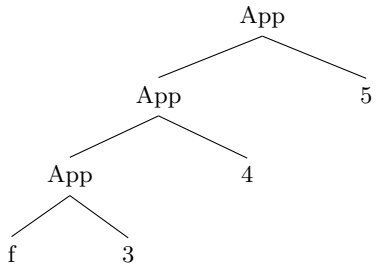


Figure 6. Core ast for the expression `f 3 4 5`

Another problem is the poor documentation of the Core datatypes in GHC and the functions which deal with it. Most function are hardly documented at all. When writing our proof-of-concept most functions used were found through a combination of patience, tracking type signatures and the help of others. Often it was necessary to check the source code of GHC to learn what a function did. To easily be able to use GHC Plugins to implement domain-specific optimisations it would help EDSL authors to have a manual for the most used functions.

9. ACKNOWLEDGEMENTS

We would like to thank Philip Hölzenspies for his patience, feedback and help in writing this paper. Further we would like to thank Marieke Huisman, Nick ten Veen and Marcel Boersma for their helpfull reviews.

10. REFERENCES

[1] New plugins work. `https://ghc.haskell.org/trac/ghc/wiki/NewPlugins`. Accessed: 05-2014.

[2] M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB is easy! In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*, pages 71–82. ACM, 2014.

[3] A. Farmer and A. Gill. A language for domain specific optimizations in Haskell.

[4] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *ACM SIGPLAN Notices*, volume 47, pages 1–12. ACM, 2012.

[5] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[6] G. Hutton and E. Meijer. Monadic parser combinators. 1996.

[7] P. T. Jager. Yard - proof-of-concept GHC plugin. `https://github.com/PimJager/Yard/`. Accessed: 06-2014.

[8] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. 2001.

[9] S. Marlow, S. Peyton Jones, et al. The Glasgow Haskell Compiler, 2004.

[10] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[11] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the Tree. In *Implementation and Application of Functional Languages*, pages 86–103. Springer, 2013.

[12] N. Scultorpe, N. Frisby, and A. Gill. The kansas university rewrite engine.