Optimising Embedded Domain Specific Languages (eDSL) using Template Haskell.

A case study on parser combinator eDSLs.

L.J. Buit
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.j.buit@student.utwente.nl

ABSTRACT

(Embedded) Domain Specific Languages (eDSL) are becoming a significant part of our programming challenges. Many programming languages have to interface with SQL, HTML, etc. A specific area of research are *parser combinator* libraries like Parsec (written in Haskell), which unlike *parser generators*, are barely optimised. This paper offers a solution to optimise a dialect of Parsec and demonstrates it with a proof-of-concept implementation. It uses Template Haskell to extract a grammar from a file during compilation and targets Happy to do the actual optimisation. The prototype only serves as an example to test how useful Template Haskell is for extracting domain knowledge (which is in our case a grammar). The intention of this research is mainly on whether Template Haskell proves a good tool to optimise eDSLs in general.

Keywords

eDSL, Template Haskell, Optimisations, Parsec, Parser Combinators

1. INTRODUCTION

A common problem in modern day software is that some other language has to be embedded in our programs. For example, many scriptable game engines use Lua [9]. Lua is a small programming language that has enough features to be useful, but is still easily embedded. Arising from the use of these so called domain specific languages (DSL) is that these cannot be optimised by the same compiler as our host language. For example, when Lua is used in a C++ program, the compiler for C++ (host language) does not know how to optimise Lua (the embedded language).

DSLs can be divided into multiple classes. In many of the game engines, Lua is a regular DSL, as no language constructs from the host language are used in the embedded language. On the contrary, embedded domain specific languages (eDSL) can exploit (among others) syntax features from the host language (examples follow in 3.1) and are therefore more tightly integrated into the host language.

1.1 Parsers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2014, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

Many programs parse data formats (e.g. JSON, XML). These files tend to get big fairly fast, so the parser has to be optimised strongly to parse these efficiently. The eDSL we study is indeed a type of parser and resembles a well known library called, Parsec [12]. Parsec is a parser combinator library, it defines some atomic parsers that can be combined using (monadic) combinators. Examples of these combinators are `<|>` (alternative) and `>>` (sequencing). Parsec is therefore quite different from the more traditional *parser generators*, which are not interleaved in the host language at all. Parser *generators* take an input (decorated) grammar, and generate a program that can parse input matching that grammar. Well known examples of parser generators are GNU Bison [1],ANTLR [13] and Happy [8]. One of the (current) benefits of using a parser generator is that most of them optimise their input.

To demonstrate the differences between Happy (first) and Parsec (second) take a look at the following example that reads "AB" or "AC". For the sake of the example only the most important bits of the files are given. The example shows the difference in input clearly. The example using Happy has its own input language whereas the example using Parsec is interleaved with Haskell.

```
rule : 'A' 'B' {}
     | 'A' 'C' {}
```

```
rule :: Parser ()
rule = (char 'A' >> char 'B') <|> (
    char 'A' >> char 'C')
```

The Parsec dialect we will be studying is more limited in its expressive power. Parsec defines many functions that make it easier to build parsers. Sadly all this extra functionality makes it harder to analyse such parsers. Our focus is on extracting its domain knowledge and not on optimising Parsec. Below is a list of atomic operations supported by this subset, these can then be combined using (monadic) combinators. This library was developed by Pim Jager [10].

```
item :: Parser Char
char :: Char -> Parser Char
digit :: Parser Char
string :: String -> Parser String
number :: Parser String
```

1.2 Extraction and Optimisations

The exact goal of this research is to analyse these programs and transform them to optimal implementations. To do so we need to first introduce domain knowledge. It was briefly mentioned before but not yet explained. We as implementor of the parser combinator library have knowledge of the parser domain. We know that a parser is always backed by a grammar to which its input is matched. The domain knowledge in this case is the grammar. For extraction of that grammar we will be using Template Haskell, a tool to alter code at compile time. Besides the obvious optimisations we want to do to the parser domain we will mainly rate Template Haskells performance in extracting that domain knowledge. We want to know if Template Haskell (an introduction is given in Section 3.4) is viable to extract domain knowledge. Once the domain knowledge is extracted we will use Happy to do the actual optimisations, Happy will remove empty rules and minimise look ahead of rules (See Section 3.2 for a visual representation). Finally Template Haskell will again be used to reintroduce the code into Haskell. Formally, we look at Template Haskells performance as a optimisation tool for eDSLs and will do so using the parser combinator library that was described before as an example.

1.3 Research Questions

- How useful is Template Haskell to give domain specific knowledge to the compiler?

1.3.1 Sub questions

- How can the domain specific knowledge be extracted from the host language? (Extract the grammar from a parser)
- How can we reintroduce the host language constructs into the extracted domain knowledge? (Extract the *decorations* from the parser)
- How can we reintroduce the optimised DSL back into the containing program. (Merge the decorated and optimised grammar back into the containing program)

1.3.2 Qualification of Template Haskell

Usefulness is a term that requires qualification. We are interested in the ease of implementation, ease of extending the program to fit more corner cases, maintainability and ease of use to the end user. It is essential that our implementation is clean and readable to cater further development in optimising eDSLs using Template Haskell.

2. RELATED WORK

Many programs depend on efficient walking of abstract data types (ADT). We will investigate an example which models a company. `Company`'s contain `Departments` which have `Employee`'s with `Salary`'s, an example company could be:

```
Company [Department [Employee 12,
    Employee 15, Employee 66], Department
     [Employee 33]]}.
```

If we want to give every employee a raise we have to write a traversal of this ADT that matches on `Employee`'s and increases their salary. Scrap Your Boilerplate (SYB) [11] is an eDSL that makes traversing ADTs easy. SYB can walk arbitrary trees without need for a new traversal boilerplate, it is therefore a *generic programming* library. SYB uses casting to see whether a function can be applied to a specific node in the tree. Its genericity is found in the combinators, for example `everywhere` is a SYB function that applies a certain transformation everywhere in a arbitrary tree.

SYB has been optimised, Template Your Boilerplate (TYB) [5] is an implementation of the same functions as SYB but using Template Haskell. The trick here is that TYB generates specific code fragments for specific ADTs, the code generated is not generic anymore and therefore TYB can match performance of handwritten code.

Another optimisation exists using Core to Core transformations [6], noted here is that SYB tries to cast values quite often. This paper uses inlining and simplification to remove these unnecessary casts.

Besides optimisations to SYB, another eDSL has been studied, Pan [14]. Pan is used to apply transformations to images. The problem with Pan programs is that it uses floating point math, colours are in range $[0, 1]$ but computers generally only have 8 bits of colour depth. This is optimised by automatically unboxing these floats to ints in range $[0, 255]$. Another interesting optimisation they do is replacing expressions with equivalent, but easier ones. For example, when an empty transformation is applied over a checker board transformation the empty transformation can be removed.

This research is unique because it applies optimisations to another eDSL, analysing parsers is very different than analysing boilerplate or Pan programs. Furthermore, we are more interested in how Template Haskell performs as an optimisation tool. Optimisations of the parser combinator library we introduced is just a way to check that performance.

3. BACKGROUND

3.1 Examples

Some examples follow, which illustrate how parser combinators can be interleaved with Haskell. The first example reads a number and, depending on whether the last character is an `L` or an `S`, makes it a `Long` or `Short`. One specific construct one should be aware of is bind (`>>=`). Bind takes a function and applies that to a wrapped value. In this example, `number` returns a `Parser String`, however we want a `Parser Numb`, so we bind the result of `number` (in this case a `String`) to a function that `read`s (converts to `Int`) that `String` and returns a `Numb`.

```
data Numb = Short Int | Long Int

rule :: Parser Numb
rule = (number >>= \x ->
          char 'S' >>
           (return $ Short $ read x ))
       <|>
       (number >>= \x ->
          char 'L' >>
           (return $ Long $ read x ))
```

Figure 1. Parser that reads a `Long` or a `Short`

This is a relatively easy program to analyse. Parser logic (`char 'S'`, `number`, `>>` and `<|>`) is hardly interleaved with Haskell constructs (λ-abstraction and `return $ Short $ read x`). Some parsers might be harder to analyse, for

example the next one, which reads a digit, followed by that *specific* amount of `a`'s. It is therefore dependent on the result of the `digit` parser.

```
rule = digit >>= \a ->
    replicateM (read [a]) item
```

3.2 Parsers as Finite State Automata

Parsers can be represented as finite state automata [15] with transitions labelled with terminals and states that define what terminals are accepted. These automata are another way of writing instances of the domain knowledge of the parser domain. We are interested in transforming non-deterministic finite automata (NFA) [15] into deterministic finite automata (DFA), or, at least, reduce the non-deterministic parts of the automata to a minimum. Take for example the grammar given in the introduction written as an NFA (Figure 2).

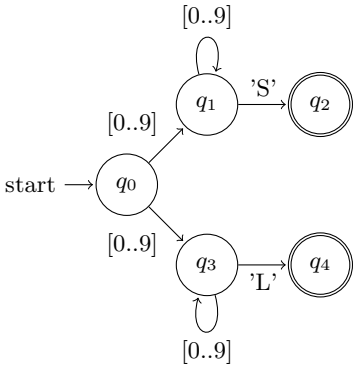

Figure 2. NFA of the Long/Short grammar

The unoptimised version of our parser combinator library will evaluate as shown in Figure 3. For the string "`1L`" first the upper path is tried, it goes with a "`1`" to q_1, but has to backtrack when no transition can be made that consumes the "`L`". Going back to q_0 and restoring the previously consumed input, it takes the transition to q_3, (again) consumes "`1`" and takes the transition to q_4, consuming "`L`". The system is now in a accepting state and therefore parsing succeeded.

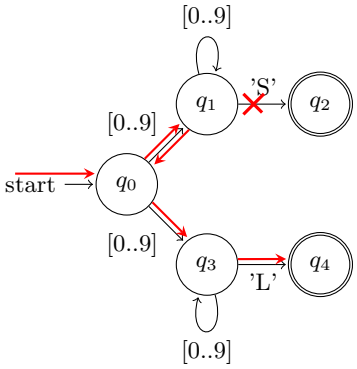

Figure 3. Traversal for input `1L`

Fortunately, this NFA can also be written as a DFA [15] by combining states q_1 and q_3 as shown in Figure 4. This is possible because both q_1 and q_3 are reached with the same input, both accept a number (consisting of multiple digits).

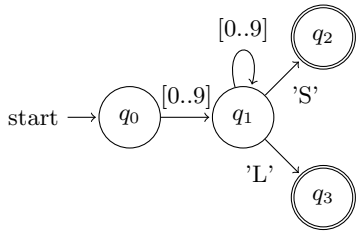

Figure 4. The NFA from figure 2 reduced to a DFA

This optimisation is exactly what needs to happen to our grammars to optimise them. Naive evaluation must try both paths when reading "`1L`" where the optimised automaton always knows what transition to take. In other words, the second automaton has a look-ahead of one. This is highly desirable because backtracking has exponential complexity, whereas a great number of languages can be parsed in linear time.

3.3 GHC

A widely used Haskell compiler is called Glasgow Haskell Compiler (GHC). It uses multiple steps to compile a source file into a binary. For further reference, a small introduction to the most relevant steps is given. A schematic representation is also given in figure 5.

Parser The parser takes the file that has to be compiled and turns it into an Abstract Syntax Tree (AST) [7]. Elements that belong together are put in the same subtree, so all subexpressions in an expression like (2+2) are arranged together. An example of a piece of AST is given in the Template Haskell section (section 3.4). The resulting AST is passed on to the renamer for further manipulation.

Renamer The renamer resolves scopes and guarantees that every unique name used is defined precisely once. This makes reasoning about variables and their definitions simpler.

Typechecker The typechecker checks the types of the program. It checks whether function calls adhere to the types of the functions etc. For example `5 + "Haskell"` will result in a type error that is reported in this compilation step.

Template Haskell Template Haskell manipulates the AST that is already renamed and typechecked. For a small reference see section 3.4. The optimisations that this paper presents are using Template Haskell.

Desugarer Transforms the decorated ASTs from the Typechecker to an intermediate language called Core [16]. This small language is easier to reason about in the optimiser that is run in the remainder of GHC.

Remainder After these steps GHC is not yet done compiling a source file, however these steps are not relevant in the context of this work. We treat the remainder of GHC as a black box that we have no control over.

3.4 Template Haskell

Template Haskell is a metaprogramming framework for Haskell. It allows programmers to generate Haskell code during compilation. The best way to think about metaprogramming is that code is run during compile time. It is for example implemented in C++ in the form of templates

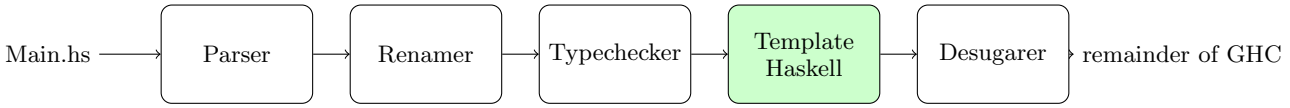

Figure 5. Compilation steps of GHC

```
template <class T>
void swapT(T& x, T& y) {
    T temp;
    temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 12;
    int b = 7;
    swapT(a,b); // Here we swap integers

    string cpp = "C++";
    string haskell = "Haskell";
    swapT(cpp, haskell); // Here we swap
        strings
}
```

Figure 6. C++ template that swaps two values

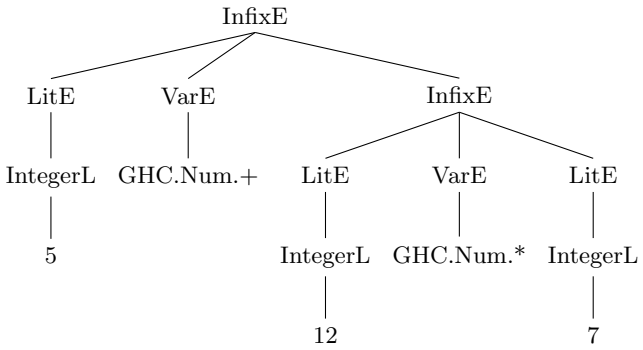

Figure 7. AST of `5 + 12 * 7`

[4]. A template is a blueprint according to which code is generated. A swap function can be defined in a generic way using such a template and when specific calls to that swap function are encountered during compilation, a specific version of that swap function for the given types is created (Adapted from [3]). Template Haskell however offers far more control over code during compile time.

3.4.1 The Template Haskell AST

The way Template Haskell allows us to do metaprogramming is by altering the AST. Figure 7 is an example of an AST that models the expression `5 + 12 * 7`. In Haskell this is represented as an ADT. The same expression is given in Figure 8 but this time in ADT format. It is interesting to note here, that both left and right side of the plus expression are wrapped in `Maybe` values, a data type with either `Nothing` or `Just` a value. It is therefore possible that the left or right side of this expression is `Nothing`. This is very useful for using partial functions. `(+5)` is also a valid Haskell expression modelling a function that takes a single argument, namely the left hand operand.

3.4.2 Splice

```
InfixE
    (Just $ LitE $ IntegerL 5)
    (VarE GHC.Num.+)
    (Just $ InfixE
            (Just $ LitE $ IntegerL 12)
            (VarE GHC.Num.*)
            (Just $ LitE $ IntegerL 7)
    )
```

Figure 8. ADT of `5 + 12 * 7`

```
-- | Select takes the ith element out of
     the n elements
select :: Int -> Int -> ExpQ
select i n = do
        x <- newName "x"
        return $ LamE [TupP (wilds x)] (
            VarE x)
    where
        wilds x = replicate (i-1) WildP
            ++ VarP x : replicate (n-i)
            WildP
-- | puts '2' onto the stdout
-- | \(_,x,_,_) -> x is spliced in.
main = putStrLn ( $(select 2 4)
    (1,2,3,4))
```

Figure 9. Select an element in a tuple

To facilitate altering the AST, Template Haskell defines two operations. The first is a splice. which takes an ADT and puts it at the current position in the file. Spliced code is also typechecked before compilation continues. An example of a splice is a function that selects an element out of a tuple. In Haskell, this is only defined for *pairs* in the form of the `fst` and `snd` function. Simulating this in Template Haskell, however is easy (Adapted from [2]). We only need to make a lambda function that takes an arbitrary length tuple, and selects an element. The select function from Figure 9 creates such a lambda function that matches all but the n^{th} element of the tuple with a wildcard pattern, while binding the n^{th} element to a variable and evaluating to that variable. A small example is given in the `main` function, with its splice (`$(..)`) expanded in the second comment.

3.4.3 Quasiquote

Quasi-quoting is (almost) the reverse operation of a splice. In a quasi-quote (`[| exp |]`) the user puts an ordinary Haskell expression that then gets translated to a Template Haskell ADT. The only difference with the splice is that a quasi-qoute does not directly alter the AST. It merely translates Haskell to that AST format. A common pattern is `$(optimise [| exp |])` where first a quasi-quote operation translates `exp` to an AST representation, then `optimise` is run on that AST and finally the result is reinserted into the program, using the splice.

```
data JSON = Object [KV] | Array [JSON] |
     Value String

data KV = KV String JSON

parseJSON :: Parser JSON
parseJSON = do
        char '{'
        kv <- many parseKeyValue
        char '}'
        return $ Object kv
    <|>
    do
        char '['
        json <- many parseJSON
        char ']'
        return $ Array json
    <|>
    parseString

parseKeyValue :: Parser KV
parseKeyValue = do
    (Value str) <- parseString
    char ':'
    json <- parseJSON
    return $ KV str json

parseString :: Parser JSON
parseString = do
    char '"'
    str <- many (satisfy isAlphaNum)
    char '"'
    return $ Value str
```

Figure 10. Example parser that parses a JSON dialect

3.4.4 Reify

Reification is the process of analysing functions. Invoking `reify` on a `Name` yields an `Info` data type containing the definition and meta-information for that name. Depending on the kind of name (variable, function, etc.) a different set of information is returned. If a function is reified, its name, type signature, list of declarations and priority (also know as fixity) is given.

4. APPROACH

To optimise our programs, we use a parser generator (Happy). Parser generators take a decorated grammar that they optimise and then implement that grammar in a Haskell. These grammars are very suitable to us because they are our domain. We know how to translate a `Parser`, consisting of parser combinators, to a grammar. This section refers to an example that parsers JSON (see Figure 10). It does not completely model the reality of parsing JSON. To keep the example compact we took shortcuts and disregarded the standard to aid in clarity.

4.1 Extracting Instances of Domain Knowledge

Extracting of domain knowledge consists of two parts. On the one hand, we have to find the code that needs optimisations and, on the other hand, we need to extract that so that our optimiser can reason with it. Firstly, we need to define what code we can reason about. As stated before, we know about atomic parsers (`item`, `digit`, etc.), and parser combinators (`<|>`, `>>=`, etc.) but thats it. A safe assumption is that all functions resulting in a `Parser a` can be interpreted as something that has meaning in our domain. All other constructs are treated as black boxes. A small nuance has to be applied here. Some functions do result in a `Parser a`, like `return`, but do not hold additional value to our domain. Finding all functions that result in a `Parser a` is solely a way to have a starting point.

Implementing this in Template Haskell should be straight forward. We should first scan a module for functions that do result in a `Parser a` and then retrieve their declarations. However, we do have a way to read functions, `reify`, but not a way to reintroduce that function with an optimised body. We cannot splice in a function that is not currently being handled by Template Haskell. Moreover we do not have a way to read an entire module at once. One would say that `reifyModule` would do that but it sadly only returns information on which modules are imported in a module, as opposed to what the function name would make us expect (give a complete representation of the module). To circumvent the issue here, we use the `$( optimise [| expr |])` construct as presented above so that we can continue optimising single expressions, we therefore moved away from analysing entire modules and resorted to optimising single functions.

To demonstrate what we want to accomplish, all functions that result in a parser are assumed to have been found. In the case of Figure 10 this is the set `parseJSON`, `parseKeyValue` and `parseString`.

4.2 Simple Analysis

The expressions that get fed to the `optimise` function are verbose. Template Haskell runs before the desugarer and therefore the entire syntax is still present. This shows, for example, in inline function application, i.e. `(>>=) a b` is differently represented in the AST then `a >>= b`. Furthermore, some constructs can be represented in a smaller subset of Haskell. An example is the `do` statement. This statement is only syntactic sugar. These ambiguities make the code harder to process, but could also point us to the way the programmer was thinking when he wrote it. We choose to remove this additional syntax, by manually desugaring to Core so that we can easily read it mechanically.

Analysis of the desugared code is then relatively easy. Functions in Core are variables that are applied to an argument. Functions that take multiple arguments are applied consecutively to single argument. Using a pattern match, we can then find out what number of arguments a function has. For example, if a piece of AST matches `AppE (AppE (VarE x)a)b`, the function `x` takes two arguments.

Functions are represented by `VarE`'s with a `Name` which (conveniently) have an instance of `Eq`. So differentiating between `>>` and `>>=` can easily be done using a comparison with the function we expect.

4.3 Simple Extraction of Grammar

For the parser example, we use Happy to optimise our eDSL. This is not fundamental to this research. We could have easily chosen to optimise using other techniques but Happy is convenient. Key in working with Happy are its grammar files. We need to somehow transform our code to a Happy grammar file. The Haskell function `parsJSON` is translated to Happy and is shown in Figure 11. Some notes have to be made here. Firstly Happy does not support `+` (one or more occurences of the previous) and `*` (zero

```
parseJSON : '{' parseManyKeyValue '}'
          | '[' parseManyJSON ']'
          | parseString

parseManyJSON : {- empty -}
              | parseManyJSON parseJSON

parseManyKeyValue : {- empty -}
                  | parseManyKeyValue
                    parseKeyValue
```

Figure 11. A Happy rule corresponding with the grammar of the Haskell function `parseJSON`

or more occurences of the previous), so these are implemented in the rules `parseManyJSON` and `parseManyKeyValue` . Secondly, this file describes a valid grammar rule in Happy. It however does not have the same behaviour as the Haskell `parseJSON` function. All constructs that did not hold domain knowledge were blatantly thrown away.

Implementing this translation is relatively easy. We just recurse the ADT and accumulate a `HappyFile` data record containing fields needed in Happy grammar files. Generally, when we encounter a '`(>>) a b`' we analyse `a` and `b` and combine these rules. If '`<|> a b`' is encountered we analyse `a` and `b` and then make an option for each. If we encounter an atomic parser (`digit`, `char`, etc.) we replace it with its associated token. All other constructs are not supported in the simple analysis.

Some care has to be taken regarding `<|>` though. It can be both a top level alternative (`rule = char 'a' <|> char 'b'`) or an 'inline' alternative (`rule = char 'a' >> (char 'b' <|> char 'c')`). These two are differently represented in Happy. We therefore first check if the first application of a function is `<|>`, if that is the case we make an alternative using its arguments in which `<|>` is treated as being inline. Besides these differences, Happy also requires us to declare all tokens that follow in the file, so during analysis also a list of tokens was accumulated.

4.4 Reintroduce Generated Code

A Happy file was generated in the previous step, which can be compiled into a Haskell module. This module defines a function that matches that token stream and results in either an error or a result. Key here however, is that this module did not exist during initial compilation steps. It got generated during the Template Haskell step. There are two key problems here. Firstly, Haskell can not find this module during its dependency search, so it does not know that it needs to be compiled. Secondly, when the module defines a function we cannot splice in a call to that function. Haskell does not know that module so it does not know its functions either.

This issue is circumvented by first creating a dummy module with the same name and (exported) functions. During compilation, we replace this module by the one created by Happy and recompile the entire program. Furthermore, the function exposed from the Happy module must also results in a '`Parser a`' so that it can be used in parser combinator expressions again.

4.5 Reintroducing Interleaved Haskell

During previous steps we removed all code that was not analysable to us. Happy luckily allows one to place code fragments into the grammars, much the same way as ANTLR. We can use this to reintroduce the previously discarted specific code. Firstly, we need to make a single distinction. Some code can be inserted right away, where other code depends on the result of a parser. The second grammar of the Background uses the result of the digit parser to read *exactly that* number of `item`s. This dependency on the outcome of a Parser is hard to model in Happy and is beyond this research. Therefore, we check that parsers do not bind (`>>=`) to other atomic parsers. The main function that needs to be identified is `return`, and frankly this is easily simulated in Happy. Further functions that are interesting are functions of `Functor`, `Applicative` and `Alternative` typeclasses. The best way to implement those is by knowing how to analyse functions of `Monad`, derive instances for `Functor`, `Applicative` and `Alternative` from that monad instance an then use an inliner to translate functions to the `Monad` functions (thus replacing functions by their definition in term of monad functions). This was an idea suggested by P.T. Jager who in fact borrowed it from P.K.F. Hölzenspies (personal communication).

5. RESULTS

A module was created that can analyse these Parsers. Using the restrictions posed in the approach, we are able to intervene in the compilation process, analyse basic constructs from that code and generate a compilable Happy file. The prototype is not able to handle interleaved Haskell, but care was taken to be able to easily implement this. Reintroducing the module into the program is also managed, but, as stated before, only when a dummy is firstly created. Small errors in the generated Happy files prevented us from further researching the performance aspect of this optimisations.

6. DISCUSSION

6.1 Viability of Template Haskell

We mainly focused on seeing whether using Template Haskell to extract, optimise and reintroduce instances of domain knowledge was a viable approach. The terms used in the research questions to measure this are discussed here. Furthermore, some notes on using Happy to optimise parser combinator libraries are given and finally some recommendations for Template Haskell are given.

6.1.1 Ease of implementation

One of the main boasts for Template Haskell is its rich syntax as it is run before the optimiser. This however has little benefit on our research. The extra syntax is a burden for us to think about and only introduces another step to compilation. Because `reify` is not completely implemented (function declarations are never returned, so no way to look inside a function) and `reifyModule` does not result in a top level view of the module, we were not able to inspect an entire module all at once. Furthermore, if we could have a view of an entire module and could have found all functions that are of interest in our domain, we still could not alter a function from within a splice in another function. Some implemented functionality clearly could have been nicer if Template Haskell provided us with correct APIs to do so. The way we introduce our optimised module to GHC, is for example a hack.

Another problem currently faced is not being able to inline code before it enters Template Haskell. The main use case was described in the approach. We could then easily derive Happy constructs for type classes like `Functor`, `Applicative` and `Alternative` when these were implemented using functions of the monad type class. Further use cases include running specific optimisations be-

fore Template Haskell to make code more manageable. Currently this is not supported. If one wants a specific optimisation from GHCs optimiser during Template Haskell, it has to be reimplemented (or at least altered) to work in Template Haskell.

6.1.2 Ease of extending to fit more corner cases

Extending the current prototype is not hard. Introducing new functions to analyse is easy if a way can be found to represent them in Happy. The data format used internally to model Happy might need to be altered to support those changes but this is manageable.

An interesting problem, however appears when two DSLs have their knowledge interleaved. What happens when `Parser`s and `OpenGL` are interleaved? Not much one would say, but when the parser first optimises its code and desugars it to Core, the (desugared) code might not be manageable for the `OpenGL` optimiser. This case however is currently not supported. Code that gets optimised by the `Parser` optimiser gets moved to a dedicated module over which the optimiser for `OpenGL` has low control.

6.1.3 Maintainability

The code currently built is maintainable in that it does not use private or deprecated functions. It is however not maintainable in the workarounds that needed to happen to make it work. These workarounds cluttered the code significantly.

6.1.4 Ease of use

This is the main problem for Template Haskell. We were not able to hide all Template Haskell code from the user. They need to manually insert code into a splice/quasiquote block (`$(optimise [| exp |])`) and no ways were found to fix this problem. This is untolerated; if end users need to manually adapt all their functions to fit a certain optimisation, it might as well be easier for them to hand-roll that optimisation in the process. Our plan was to adapt the `runParser` function so that when the user tried running its parser we would have replaced it with the one optimised. Adapting `runParser` was, however, infeasable because altering the parser functions from within proved to be impossible.

6.2 Viability of Happy

Happy is not very suitable to analyse code written using a parser combinator library. Although it was no real objective to this research, it is noteworthy. The grammars Happy accepts as input are not flexible enough to allow for all constructs possible. More importantly, one would easier be tempted to use these constructs when using with a monadic parser combinator library. Furthermore, but unsurprisingly, using external tools during development of optimisers is highly discouraged. Because Happy generates new modules at compile time, we encountered a batch of problems when reintroducing that code in Haskell.

6.3 Recommendations to Template Haskell

For Template Haskell to be useful in these optimisations, a clear way is needed to introduce or alter other functions from within a splice. Furthermore, the full implementation of `reify` could be very useful when we, for example, want to inline certain functions from within a splice. It is also beneficial for `reifyModule` to return a list of top level declarations of a module, so that we can find all functions that have meaning in our parser domain. Finally, it would be nice if it was possible to introduce new modules to the compiler when compilation happens. We would then need to be able to insert imports in the module that is currently being processed by Template Haskell. How feasible all these recommendations are is to be decided by the Template Haskell maintainers. For this use case we would greatly benefit from those functions.

7. CONCLUSIONS

Template Haskell is not very viable for optimisations of DSLs in Haskell. Its rich syntax is a burden to analyse and its lacks ways to introduce new modules into the compile tree. Also because Template Haskell is run before the desugarer, we cannot invoke optimisation passes that GHC already knows. The benefits of Template Haskell do not compare to its downsides. Its ways to generate fresh names and easily reference already existing unique names are not much needed.

8. FUTURE WORK

Besides the additional features to Template Haskell we can clearly improve on these optimisations too. This case study used external tooling to optimise grammars, but this is not ideal. Happy is not made for parser combinators. Tricks used when using parser combinator libraries are not always easy to emulate in Happy files. If a complete optimisation of parsers is needed it is perhaps better to build an own representation with more expressive power so that dependencies between parser output and input can be described better. A good starting point would be the finite automata introduced in the background, when combined with ways to keep track of variables being passed to next parsers it could very much be a better representation of parser combinators than we can achieve using the more traditional parser generators.

9. ACKNOWLEDGEMENTS

Philip Hölzenspies for his countless hours of guidance and (among others) the idea of using an inliner for easily analysing `Functor`, `Applicative` and `Alternative`.

10. REFERENCES

[1] GNU Bison, description of the project. http://www.gnu.org/software/bison/. Accessed: 09-03-2014.

[2] Haskell wiki page for template haskell. http://www.haskell.org/haskellwiki/Template_Haskell. Accessed: 12-06-2014.

[3] Implementation of swap using C++ templates. http://www.cplusplus.com/reference/algorithm/swap/. Accessed: 13-06-2014.

[4] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[5] M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using template haskell for efficient generic programming. pages 13–24, 2012.

[6] 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*, PEPM '14, pages 71–82, New York, NY, USA, 2014. ACM.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 1988.

[8] A. Gill and S. Marlow. Happy: the parser generator for haskell. *University of Glasgow*, 1995.

[9] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Reference manual of the

programming language lua. *Monografias em Ciências da Computação*, 4:94, 1994.

[10] P. T. Jager. Analysing embedded domain specific languages in haskell from core. 2014.

[11] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ACM SIGPLAN Notices*, volume 38, pages 26–37. ACM, 2003.

[12] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[13] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[14] S. Seefried, M. Chakravarty, and G. Keller. Optimising embedded dsls using template haskell. In *Generative Programming and Component Engineering*, pages 186–205. Springer, 2004.

[15] T. A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[16] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.