Embedded parser generators

Jonas Duregård Patrik Jansson

Chalmers University of Technology and
University of Gothenburg
{jonas.duregard,patrikj}@chalmers.se

Abstract

We present a novel method of embedding context-free grammars in Haskell, and to automatically generate parsers and pretty-printers from them. We have implemented this method in a library called BNFC-meta (from the BNF Converter, which it is built on). The library builds compiler front ends using metaprogramming instead of conventional code generation. Parsers are built from labelled BNF grammars that are defined directly in Haskell modules. Our solution combines features of parser generators (static grammar checks, a highly specialised grammar DSL) and adds several features that are otherwise exclusive to combinatory libraries such as the ability to reuse, parameterise and generate grammars inside Haskell.

To allow writing grammars in concrete syntax, BNFC-meta provides a *quasi-quoter* that can parse grammars (embedded in Haskell files) at compile time and use metaprogramming to replace them with their abstract syntax. We also *generate* quasi-quoters so that the languages we define with BNFC-meta can be embedded in the same way. With a minimal change to the grammar, we support adding *anti-quotation* to the generated quasi-quoters, which allows users of the defined language to mix concrete and abstract syntax almost seamlessly. Unlike previous methods of achieving anti-quotation, the method used by BNFC-meta is simple, efficient and avoids polluting the abstract syntax types.

Keywords Metaprogramming, Grammar Engineering, Domain Specific Languages, Quasi-quoting

1. Introduction

The underlying motivation of this paper is to support rapid development of, and experimentation with, Domain Specific Languages (DSLs). Especially if the desired syntax of the DSL makes it difficult or impossible to embed it in a general purpose language using conventional methods (as a combinator library). We aim to eliminate the "barrier" associated with employing a parser generator such as BNFC, and make it as easy to use as a parser combinator library but provide a wider set of features.

The title of this paper is deliberately ambiguous about what is embedded, referring both to the parser generators and the generated parsers.

Embedded (parser generators) Like the original BNF Converter on which it is built, BNFC-meta builds compiler front ends (abstract syntax types, parsers, lexers and pretty-printers) from grammar descriptions. Unlike the BNF Converter and other parser generators: 1) BNFC-meta is a metaprogram and 2) our grammar descriptions are embedded as Haskell values.

By "metaprogram" we mean that it is a Haskell function from grammars to abstract Haskell code. This abstract code can be "spliced" by the compiler using the Template Haskell language extension. Shortening the compilation tool chain in this way has many practical advantages, including faster and simpler compilation. Embedding the language definitions also allow users to define their own functions in the same module as the parser.

The fact that grammars are embedded in the target language (i.e. Haskell) is a major advantage of our approach. This lends BNFC-meta features which are typically reserved for combinator parsers, namely the possibility of building grammar definitions using all the abstraction features of a functional language. This can drastically reduce the complexity of code, enabling features such as reusable grammars, parameterised grammars and programmer defined grammar transformations. Even though BNFC-meta grammars are embedded in Haskell rather than defined in separate files, BNFC-meta allows users to write grammars in the same concrete syntax as BNFC does. This is achieved using another metaprogramming facility called quasi-quoting, which essentially provides programmer defined syntax extensions to Haskell. Users can mix concrete and abstract grammar syntax depending on which is most suited for the task at hand.

(Embedded parser) generators The alternative interpretation of the title (that the generated parsers are embedded) highlights another innovation in BNFC-meta. By embedded we mean that any language defined with BNFC-meta (an object language) can be used as a syntactic extension to Haskell, and the compiler will automatically translate the concrete syntax of the object language into its corresponding abstract syntax. This is achieved using the same technique as when we embed our grammar definitions; we automatically generate quoters for the object language. As indicated by their name, a quoter allow the object language to be used in a syntactically defined scope (a quote).

Quotes can be used to define both patterns and expression, but in order to define useful patterns we need to be able to bind variables. In general we want to be able to have "holes" in our quotes that are filled with Haskell values. This is called anti-quoting, and it is what separates a *quasi*-quoter from just a quoter. In BNFC-meta there is built in support for defining anti-quotation.

The embedded parsers generated by BNFC-meta enables the programmer to "mix and match" abstract and concrete syntax of the object language seamlessly, reducing code size and increasing readability for several common language processing tasks.

2. Examples

The remainder of this paper is structured as follows. In this section we present the tools on which BNFC-meta is built and some examples of using it. In Section 3 we present the details of embedding BNFC into a Haskell library. We also argue that this method is not only useful for BNFC, but can be applied in several other contexts. In Section 4 we explain the quasi-quoting mechanisms generated by BNFC-meta, and explain why these are a natural consequence of embedding BNFC as a library. In Section 5 we show some performance results and a large scale example. We conclude in Section 6 with a discussion of related and future work.

2.1 Preliminaries

BNF, LBNF and BNFC The Backus-Naur Form (BNF) is a notation for context free grammars. The notation is widely used in computer language processing, mainly due to the fact that efficient parsers can be automatically constructed from BNF grammars. There are many variants of BNF but all of them have production rules from which you can build sentences:

$$\begin{aligned} Foo &::= \texttt{"Foo!"}\ Foo \\ &\mid Bar \\ Bar &::= \texttt{"Bar."} \end{aligned}$$

The category Foo (also called a nonterminal, as opposed to the terminal strings) represents all sentences on the form "Foo!Foo! ... Foo!Bar.". We use a variant of BNF called Labeled BNF (LBNF). In LBNF each production rule represents a single choice (there is no vertical bar operator) and each production rule carries a descriptive label. The grammar above can be expressed as:

$$\begin{aligned} &FooCons.\ Foo ::= \texttt{"Foo!"}\ Foo; \\ &FooNill.\ \ Foo ::= Bar; \\ &BarDot.\ \ Bar ::= \texttt{"Bar."}; \end{aligned}$$

The primary advantage of LBNF is that a system of algebraic data types can be extracted from the rules by using categories as types and labels as constructors. These types capture the abstract syntax tree of the object language. In our example:

$$\begin{aligned} &\mathbf{data}\ Foo = FooCons\ Foo \mid FooNill\ Bar \\ &\mathbf{data}\ Bar = BarDot \end{aligned}$$

In LBNF users can add custom lexical tokens to the language by specifying a regular expression. Each token is given a name, and constitutes an independent grammar category which is represented in the abstract syntax by a **newtype** wrapper around strings. In BNFC-meta we offer a slightly generalised version of these token definitions where each token has a category and a label (just like any other rule). We write:

$$FoosToken.\ Foos ::= @(\texttt{'F'}\ \texttt{'o'}\ \texttt{'o'}\ \texttt{'!'})*$$

The @ just denotes that this is a token definition. There are more compact ways of writing this particular expression. The abstract syntax type generated by this example is simply $\mathbf{data}\ Foos = FoosToken\ String$. Strings matching the regular expressions are wrapped by the constructor ($FoosToken$) representing the defined rule.

The BNF Converter (BNFC) is a program that uses LBNF grammars to generate complete compiler front ends. This includes abstract syntax tree types, a lexical analyser (lexer), a parser and a pretty printer. BNFC is written in Haskell but can generate parser code for many target languages, including Haskell, C/C++ and Java. BNFC-meta does not support any target language other than Haskell, and when we refer to the BNFC in the remainder of this paper we actually mean only its Haskell back end. The LBNF grammar formalism has many features which are not described in this paper (see [Forsberg and Ranta 2003] for details).

Template Haskell and Quasi-quoters Template Haskell is a metaprogramming extension to Haskell, first described by [Sheard and Jones 2002]. The features of Template Haskell relevant to this paper include:

- A library of data types for the abstract syntax of Haskell, including types for declarations, expressions, patterns etc. We call these values metaprograms.
- A language extension for "splicing" the metaprograms into Haskell source code at compile time.

Suppose we have defined a few metaprograms (in this case simple code fragments) in a module $MyTemplates$. With a recent version of Template Haskell the following program is possible:

$$\begin{aligned} &\{\text{-\# LANGUAGE TemplateHaskell \#-}\} \\ &\mathbf{import}\ Language.Haskell.TH\ (Q, Dec, Exp) \\ &\mathbf{import}\ MyTemplates\ (myDeclarations, \quad \text{-- :: Q [Dec]} \\ &\qquad\qquad\qquad\qquad\quad myExpression) \quad \text{-- :: Q Exp} \\ &\quad \text{-- Top-level Haskell declaration splice} \\ &myDeclarations \\ &\quad \text{-- Expression splice} \\ &splicedExpression = \$(myExpression) \end{aligned}$$

In this example the expression $myDeclarations$ is used in place of a sequence of declarations at the top level. The compiler will evaluate it, splice in the resulting declarations in its place and continue to compile the resulting code. In expression contexts splices must be indicated by a $ but the idea is the same: evaluate, splice in, recompile. Note that the Q monad can perform IO actions so even this simple example could potentially splice "dynamic" code e.g. by reading a description from a file or base the particular code on the architecture of the compiling machine.

Quasi-quotes The term quasi-quotation is not used entirely consistently in the functional programming community. In a broader linguistic setting, the term quasi-quote was coined by W. V. Quine in 1940 as a means of formally distinguishing metavariables from surrounding text. In computer languages such as Lisp, the term quasi-quote is almost synonymous with template metaprogram i.e. a metaprogram that may contain "holes" for the programmer to fill. This type of quasi-quotes are also in the original Template Haskell design. Understanding exactly how this kind of quasi-quoter works is not essential for this paper, but we show a (somewhat contrived) example of how it can be used:

$$\begin{aligned} &sharedPair :: Q\ Exp \to Q\ Exp \\ &sharedPair\ e = [|\ \mathbf{let}\ x = \$e\ \mathbf{in}\ (x, x)\ |] \end{aligned}$$

The [| initiates the quote, which means that the Haskell code within it is a metaprogram. The dollar sign marks holes in the metaprogram, where e is the the parameter of $sharedPair$. The general term for the dollar sign is *anti-quotation* operator, since it escapes from the quoted context into the surrounding metalanguage.

The QuasiQuotes extension to Haskell, introduced in [Mainland 2007], offers a generalisation of quasi-quotes where any *object language* can reside in a quote. The Lisp and Template Haskell quasi-quotes are the special case where the object language is the same as the enclosing language. The translation of the object language into Haskell is defined by the user (by writing a parser). Each quote is labeled with the name of a quasi-quoter (defined in another Haskell module) which is used to parse the specific quote. In other words a Haskell quasi-quoter is essentially syntactic sugar for metaprograms parameterised by a string. If q is quasi-quoter, then we can write the declaration $x = [\,q \mid \phi \mid]$ and the compiler applies q to the $String$ containing the text ϕ. This produces a metaprogram of type $Q\ Exp$ which is spliced by Template Haskell, replacing the quasi-quote. Note that this generalized type of quoters does not have a

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}                    1
import Language.LBNF                                            2
bnfc [lbnf |                                                    3
RAlt.   Reg1 ::= Reg1 "|" Reg2;                                 4
RSeq.   Reg2 ::= Reg2 Reg3;                                     5
RStar.  Reg3 ::= Reg3 "*";                                      6
REps.   Reg3 ::= "eps";                                         7
RChar.  Reg3 ::= Char;                                          8
_.      Reg  ::= Reg1;                                          9
_.      Reg1 ::= Reg2;                                          10
_.      Reg2 ::= Reg3;                                          11
_.      Reg3 ::= "(" Reg ")";                                   12
|]                                                              13
example :: Reg                                                  14
example = RStar (RAlt (RChar 'a') (RChar 'b'))                  15
```

Figure 1. Basic usage of the *Language.LBNF* module.

universal anti-quotation operator like the built in Template Haskell quasi-quoter does. Instead the programmer of each quasi-quoter must define the syntax for anti-quotation and the proper translation of the anti-quoted text into Haskell expressions and patterns. Many Haskell quasi-quoters don't support anti-quotation at all, meaning they are are really only quoters (and not actually "quasi"). In the rest of this paper we will still use the term quasi-quoters to refer to any quoters regardless their anti-quotation support.

2.2 Running example

In this section we demonstrate the use of BNFC-meta, and explain how this differs from the original BNF Converter. Our object language represents regular expressions (it is actually a subset of the regular expression syntax of LBNF). Henceforth we refer to this language as *Reg* and it is used as a running example in the rest of the paper. The language has six syntactic constructs: choice, concatenation, repetition (Kleene star), empty string (epsilon), single characters and parentheses. These constructs all reside in a single grammar category, that we also call *Reg*. Since there are infix binary operators (choice and concatenation) we need to indicate precedence and associativity. In LBNF this is done by using *indexed categories*. Figure 1 (lines 4–12) shows the grammar of *Reg*, divided into three levels of precedence. We label the first five syntactic constructs (*RAlt*, *RSeq*, *RStar*, *REps* and *RChar*). The other rules (including the parenthesis rule) have no semantic importance and are not labeled; an underscore is placed instead of a label to indicate this.

Note that in Figure 1, we have embedded the grammar into a Haskell module using a quasi-quoter (named *lbnf*) and the value produced by the quoter is passed to the metaprogram *bnfc*. The end result is that the code produced by *bnfc* is spliced by the compiler, replacing the grammar definition. The details of this embedding are covered in Section 3. With the original BNF Converter, we would need to put the grammar in a `.cf` file and run the BNFC tool instead. Doing so would produce a lexer module, a parser module, a printing module and an abstract syntax module (each module in a separate output file). The module we have defined, on the other hand, can be loaded into the GHC interpreter just like any other module and the generated tools for *Reg* are readily available:

```
Ok, modules loaded: Main.
*Main> :i Reg
data Reg = RAlt Reg Reg | RSeq Reg Reg
         | RStar Reg | REps | RChar Char
instance Eq Reg
instance Ord Reg
instance Show Reg
instance Print Reg
```

This is the abstract syntax type extracted by BNFC-meta. Note that all the indices are ignored for the purpose of building the AST types, so the different levels of precedence are not reflected here. As the figure shows we can also write code that uses this abstract syntax directly in the module (like we do in *example*). The *Print* instance for *Reg* (indicated above) provides a pretty printer:

```
*Main> printTree example
"('a' | 'b')*"
```

There is also a lexer for the grammar and a parser for each category. The name of of the parser is the name of the category prefixed by the letter p and the name of the lexer is *tokens*:

```
*Main> pReg (tokens "'a' | 'b' *")
Ok (RAlt (RChar 'a') (RStar (RChar 'b')))
```

In Section 3.1 we show more advantages of embedding BNFC in Haskell.

2.3 Quasi-quoters for free

Apart from the embedding itself, the major addition in BNFC-meta compared to the original BNFC is the automatic generation of quasi-quoters. This feature generates a quasi-quoter for each category of the grammar (the name of the quasi-quoter is the name of the category, but with initial lowercase). Abstract values in the language may thus be specified using the concrete syntax. Because of the Template Haskell stage restrictions, we can not use the quasi-quoters directly in the same module. But if we import the module in which we defined the grammar we can write code like this:

```
r1, r2 :: Reg
r1 = [reg | 'a' * 'b' * 'c'* |]
r2 = RSeq (RSeq (RStar (RChar 'a'))
                (RStar (RChar 'b')))
          (RStar (RChar 'c'))
```

Here r_1 and r_2 are exactly equal, but r_1 is defined using concrete syntax whereas r_2 uses abstract. Note that the parsing of the regular expression syntax occurs at compile time, so there is no run time overhead and no risk of run time errors. Any closed expression or pattern of the type *Reg* can be expressed using these quotes.

To express patterns with variable bindings, or expressions that use variables, there needs to be a facility to escape from the quote back to Haskell. In other words we need to extend the quasi-quoters with anti-quoting. To add a simple form of anti-quoting to *Reg* we need to determine a syntax which does not clash with our other syntactic constructs. For instance we can use $\%x$ where x is an identifier bound in the Haskell environment in which the quote is situated. So we can have definitions like these:

```
plus :: Reg → Reg
plus r = [reg | %r %r*|]

xs :: Reg
xs = plus [reg | 'a' |]
```

where xs evaluates to the abstract syntax of the *Reg* expression `'a' 'a'*`. To express this kind of anti-quotation, we introduce a special label for grammar rules in BNFC-meta: $. We can use this together with the built in *Ident* category[1] to add anti-quotation to *Reg*:

[1] *Ident* is a predefined LBNF-category for identifiers. We could also roll our own identifiers or use any other category to syntactically define allowed Haskell expressions.

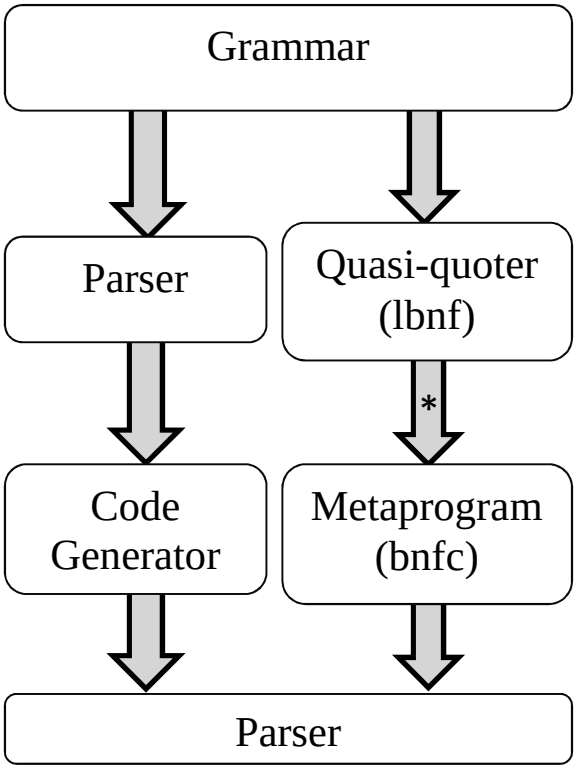

Figure 2. A comparative overview of BNFC and BNFC-meta. The asterisk indicates programmer defined transformations.

$. Reg_3 ::= "%" $Ident$;

This adds exactly the anti-quotation mode we want to the Reg parser and the example above works as intended. Replacing the label with the dollar sign indicates to BNFC-meta that this is an anti-quotation rule. By default BNFC-meta allows only one category on the right hand side of such rules. When the quasi-quoter parses using this rule, the concrete syntax of the $Ident$ is sent to a Haskell parser and the resulting expression (or pattern) is used instead of an abstract syntax expression. Adding this anti-quotation does not change the parser $pReg$ nor does it change the abstract syntax type Reg.

3. Embedding BNFC

This section discusses the process of elevating BNFC from a command line tool into a metaprogram, and the advantages of doing so. We also show that the process can be generalised and applied to any tool that satisfies certain criteria. Figure 2 gives an overview of the basic changes in BNFC-meta compared to BNFC. The benefits of embedding BNFC into Haskell are:

- Libraries that use BNFC-meta do not depend on any installed applications other than the Haskell compiler, hence they do not require custom install scripts. An alternative way to achieve this is to include the BNFC-generated code in the library; but that is generally bad practice since it makes the source more difficult to understand and it invites collaborators to edit the generated code directly, as opposed to properly changing the source grammar.
- Like with any other embedding, the features of the host language (Haskell) are available to the embedded one. Since the $Grammar$ type is exported by $Language.LBNF$ it is possible to write functions that combine or otherwise manipulate/analyse grammars at compile time, before they are spliced by the $bnfc$ function.

While the importance of the first feature should not be underestimated, it has no groundbreaking impact on the process of defining languages. The second one has more substantial benefits and is the focus of the remainder of this section.

3.1 Parser generators and combinators

Traditionally, parser generators are the dominating software tools for defining parsers. Most parser generators use some domain specific language for defining grammars, and from such definitions they produce a much more verbose parser implementation in a particular language (the target language). Sometimes a parser generator can use the same grammar to generate parsers in several different target languages. Parser generators can also generate other useful language tools such as pretty-printers and abstract syntax tree types, if the grammar DSL carries enough information [Forsberg and Ranta 2003; Ranta 2004].

In functional programming there is a compelling alternative to parser generators, known as parser combinators. Here the grammar DSL is embedded in the target language, using ordinary language features such as higher order functions to construct parsers by combining simpler ones. Major advantages over parser generators include the familiarity to the programmer (the parser is written in the same language as the code that uses it) and the comfort with which parsers can be developed (no need to generate code between test-runs). Also all the features of the target language are available when defining the parsers so general patterns for eliminating code duplication can be applied, which might not be possible in a standalone grammar DSL. But parser combinators also have several downsides compared to parser generators:

- A parser generator (especially one that is limited to context free object languages) can analyse grammars and detect ambiguities and other anomalies statically. With parser combinators these errors are either detected at runtime, or not detected at all (resulting in unpredictable behaviour).
- Combinator parsing has generally lower performance compared to parser generators, often relying heavily on backtracking or requiring the user to perform manual optimisations.
- Since combinator libraries usually rely on the recursion mechanism of the target language, they typically can't deal with left recursive grammars (which causes an infinite expansion and subsequently failure to terminate). Most grammars for real object languages rely on left recursion at some point, but any grammar with left recursion can be rewritten into an equivalent form without it [Moore 2000]. The rewriting might not be obvious though, and may impair the readability of the grammar considerably.
- Sometimes parser combinator libraries have a significant syntactic overhead compared to the streamlined grammar DSLs used by parser generators.

While BNFC-meta is definitely a parser generator (the parsers are generated Haskell code), it has several of the advantages of parser combinators. Like parser combinators it is available as a library instead of an application, thus it does not require any other tools than the compiler. More importantly, grammars are *almost* first class citizens of the target language (Haskell). We write almost first class because the grammars are only values at compile time, and sometimes this kind of values are referred to as second class citizens. This means that users can apply arbitrary Haskell functions to the grammar after parsing it with $lbnf$ but before splicing it with $bnfc$.

```
module RegexGrammars where
import Language.LBNF
import Language.LBNF.Grammar
combine :: Grammar → Grammar → Grammar
combine (Grammar i) (Grammar j) = Grammar (i ++ j)
minimal, extended :: Grammar
minimal = [lbnf |
  RAlt.   Reg1 ::= Reg1 "|" Reg2;
  RSeq.   Reg2 ::= Reg2 Reg3;
  RStar.  Reg3 ::= Reg3 "*";
  REps.   Reg3 ::= "eps";
  RChar.  Reg3 ::= Char;
  _.Reg  ::= Reg1;
  _.Reg1 ::= Reg2;
  _.Reg2 ::= Reg3;
  _.Reg3 ::= "(" Reg ")";
|]
extended = combine minimal [lbnf |
  RPlus.  Reg3 ::= Reg3 "+";
  ROpt.   Reg3 ::= Reg3 "?";
|]
```

Figure 3. Grammar reuse.

In Figure 2 this is indicated by the asterisk symbol in the flow chart. In practice this means you can write code like this:

```
import OtherModule (f)   -- f :: Grammar → Grammar
bnfc (f [lbnf | Γ |])
```

The Template Haskell stage restrictions prevent f from being defined (at top level) in the same module as the splice. Note that f is evaluated at compile time so if it fails then a compile time error is raised (see Section 4.3 for a discussion on error messages). In this case the type of f indicates that it is a grammar transformer but one could also have functions that do not take an existing grammar as an argument (i.e. constant grammars) or one that takes some other parameter like a list of operators and constructs grammar from that (i.e. parameterised grammars).

Example: grammar reuse In the original BNF Converter, the only way to extend an existing grammar is to copy the file and add the required rules. This procedure is very pervasive from a maintenance perspective. In BNFC-meta on the other hand, grammars are Haskell values (at compile time) and can thus be manipulated using all the features of Haskell (including a rich module system).

Suppose that we want to create an extended version of our *Reg* grammar (from Figure 1) that has the postfix operators + (non empty repetition) and ? (optional). First we factor the grammar out of the module in which it is defined into a new module. Then, instead of applying *bnfc* to the grammar, we give it a top level name. Thus *bnfc* [*lbnf* | Γ |] becomes *minimal* = [*lbnf* | Γ |]. The type of *minimal* is *Grammar* which is just a wrapper around a list of grammar rules. Figure 3 demonstrates how we exploit this fact to create a crude function for combining two grammars, and how this function is used to define the extended grammar in terms of the original one.

The extended parser is defined by importing *RegexGrammars* and splicing in *bnfc extended* (or *bnfc minimal* for the original language). In Section 4 we will come back to the *combine* function to add anti-quotation support to our extended grammar (see Figure 7).

3.2 A general method for embedding compilers

The principle behind the embedding is remarkably simple. Like most compilers BNFC has two distinct components:

- A front end, roughly corresponding to a function of type *String* → *Grammar*, where the string is the grammar written by the user (the concrete syntax) which is parsed into a value of type Grammar (an abstract syntax tree).
- Several back ends, each producing a parser written in a specific programming language. The Haskell back end can (somewhat simplified) be thought of as a function *Grammar* → *String* which takes the grammar of the parser and produces concrete Haskell syntax.

Conveniently, the front end corresponds exactly to the *lbnf* quasi-quoter. All that is needed to construct *lbnf* is a function that converts a *Grammar* into a Haskell expression of type *ExpQ*, such that the expression evaluates to the given grammar value. The function is trivial but a bit verbose. One method of doing this automatically for any type (using generic programming) is presented by [Mainland 2007].

Likewise, the back end corresponds exactly to the *bnfc* function. The only difference is that BNFC produces concrete Haskell syntax (*String*) whereas we need abstract syntax (*DecsQ*). The quickest way of coding this conversion from concrete to abstract syntax is to plug in a Haskell parser. A more elegant way might be to alter the BNFC back end.

This simple technique can be generalised to embed any compiler under the conditions that 1) the target language of the compiler is Haskell and 2) the compiler is implemented in Haskell. The first requirement enables Template Haskell to splice the resulting code into a Haskell module. The second requirement means that the code from the original application can be put directly into a library. Note that it is not required to have separated front / back-ends. If the application only provides a function α :: *String* → *String* we can just consider the abstract syntax tree type to be *String*. The front end quoter may then be built from the identity function, and the back end may be built from α.

Embedding BNFC and friends In practice it proved complicated to use the general method for BNFC because the back end does not only produce Haskell code. BNFC also produces intermediate code for the Happy parser generator and the Alex lexer generator, violating the first condition for applying our method. When using BNFC this means that after processing your grammar with BNFC, you will need to process the output with Happy and Alex. This is not desirable in a library setting, especially since avoiding external software is one of the perks of the embedding.

The generality of our approach enabled us to overcome this problem, by embedding Alex and Happy in much the same way as we embedded BNFC. Alex and Happy 1) both produce Haskell code and 2) are both written in Haskell, so they satisfy the criteria for embedding. The result of applying the embedding method to these programs are two new libraries: happy-meta and alex-meta. Similar to BNFC-meta, these libraries allow users to write Happy / Alex syntax directly into Haskell modules using quasi-quoters, and splice the result into the module using Template Haskell. As a byproduct, they provide the functions necessary to adapt BNFC to the first condition (by composing the back end of BNFC with the front ends of Alex/Happy). Although we used the "quick and dirty" approach (parsing code we have generated ourselves) this is only a performance issue at compile time, the produced code is essentially identical to the code produced by the original tools. We have made the three resulting libraries (BNFC-meta, happy-meta and alex-meta) available through Hackage.

```
import Regex
r1 = [reg | 'a'*'b'*'c'* |]
r2 = RSeq (RSeq (RStar (RChar 'a'))
                (RStar (RChar 'b')))
          (RStar (RChar 'c'))
isEps1 [reg | eps |] = True
isEps1 _             = False
isEps2 REps = True
isEps2 _    = False
```

Figure 4. Using simple automatically generated quasi-quoters. The values r_1 and r_2 are equivalent.

4. Quasi-quoters for free

When designing an embedded domain specific language, programmers usually have some ideal concrete syntax in mind (often something inspired by the domain in question). Conventional embedding techniques typically employ combinators and infix operators to make the interface in the host language as similar as possible to the ideal syntax. Although Haskell's syntax is remarkably flexible for embedding languages, there are restrictions on what you can do, and combinators are not useful when pattern matching.

These problems are avoided by employing quasi-quoters. Users can then insert program fragments written in the concrete syntax of the object language, and the compiler will translate these into Haskell expressions or patterns. Constructing basic quasi-quoters require little more effort than writing the parser. Since BNFC-meta already generates the parsers, it is natural to extend it to generate quasi-quoters as well.

In Figure 4 two functions are defined with and without quasi-quoters. The basic quasi-quoters are quite useful for defining constant (closed) expressions, with r_1 being considerably shorter and easier to read than r_2. For patterns the basic quoters are not very useful since they cannot bind variables. It is difficult to think of a useful constant pattern which is more advanced than the one in $isEps_1$. In fact we cannot even make a corresponding function that checks if the argument is a choice ($RAlt$) expression because we would need to bind variables which we cannot express in our Reg language. In Section 4.2 we show a solution to this problem.

4.1 Where do quoters come from?

This part of the paper will describe the technical aspects of generating Haskell quasi-quoters automatically. All quasi-quoters, like reg and $lbnf$, are of the type $QuasiQuoter$ which is a record type that contains at least an expression and a pattern quoter (recent versions of the Template Haskell library also include type and declaration quoters).

```
data QuasiQuoter = QuasiQuoter
  { quoteExp :: String → Q Exp
  , quotePat :: String → Q Pat
  }
```

The Exp and Pat types are used for building Haskell expression and patterns respectively. So the quote $[q \mid \phi \mid]$ splices the expression produced by $quoteExp\ q\ \phi$ if the quote is in an expression context and $quotePat\ q\ \phi$ in a pattern context. For the basic quoters we can restrict ourselves to this tiny subset of the Template Haskell library API:

```
mkName :: String → Name

data Exp
  = ConE Name        -- Constructors
  | AppE Exp Exp     -- Application
  | LitE Lit
  ...
data Pat
  = ConP Name [Pat]  -- Constructor application
  | LitP Lit
  ...
data Lit = CharL Char
```

From these definitions we can simply translate a parsed value into an expression that evaluates back to the given value, or a pattern that matches only the given value. For instance `"'a'"` is parsed to `RChar 'a'`, which as an expression is

$$AppE\ (ConE\ (mkName\ \texttt{"RChar"}))\ (LitE\ (CharL\ \texttt{'a'}))$$

and as a pattern

$$ConP\ (mkName\ \texttt{"RChar"})\ [LitP\ (CharL\ \texttt{'a'})]$$

This translation is completely mechanical and there are tools for performing it automatically, either using generic programming or metaprogramming. This is essentially the technique BNFC-meta uses to build basic quasi-quoters.

Bootstrapping The original BNF Converter is bootstrapped, it "eats its own dog food". Specifically it is used to generate its own front end: the syntax of LBNF is specified as an LBNF grammar, and when the grammar is fed to BNFC the output is identical to the front end of BNFC. The addition of automatically generated quasi-quoters preserves this property[2]: the quasi-quoter $lbnf$ can be automatically generated by processing the grammar for LBNF in BNFC-meta. Just like BNFC can generate a complete front end for a traditional compiler, BNFC-meta can generate a front end for an embedded one. Figure 5 shows a comparison between the components of BNFC and BNFC-meta, and the dotted lines indicate the bootstrapping capacity of each.

4.2 Anti-quoting

There is no standardised mode for anti-quotation in Haskell quasi-quoters (and many quasi-quoters do not use anti-quotation at all). Since a quoter is essentially a function $String \to Q\ Exp$ (if used in an expression context) this function can choose to treat some part of the input string as Haskell code, and parse it as such. Usually a special token followed by a single identifier is used, and the identifier is translated into a Haskell variable instead of a member of the abstract syntax type. By using **let** or **where** bindings the single identifier can represent any expression, including nested quotes (although this does not work in pattern context).

Usually quasi-quoters are built from regular parsers, i.e. from functions like $\alpha :: String \to Maybe\ \Delta$ where Δ is some abstract syntax tree type. The Template Haskell library contains functions that use generic programming to automatically construct quoters from this type of functions, and users can add the exceptions for anti-quoted code in a reasonably simple manner. The problem with this approach is that the parser must still be changed to accommodate the anti-quoting syntax. Although the changes to the parser can be mitigated by using flags to indicate if the parser is used for regular parsing or quasi-quoting (and thus enable/disable anti-quotation syntax), the abstract syntax type Δ will need to be extended to contain arbitrary strings.

[2] The concept of bootstrapping is slightly misleading in a library context, since libraries cannot depend on previous versions of themselves. In BNFC-meta this is solved by adding a small bootstrapping utility that flushes the produced code into a file instead of splicing it.

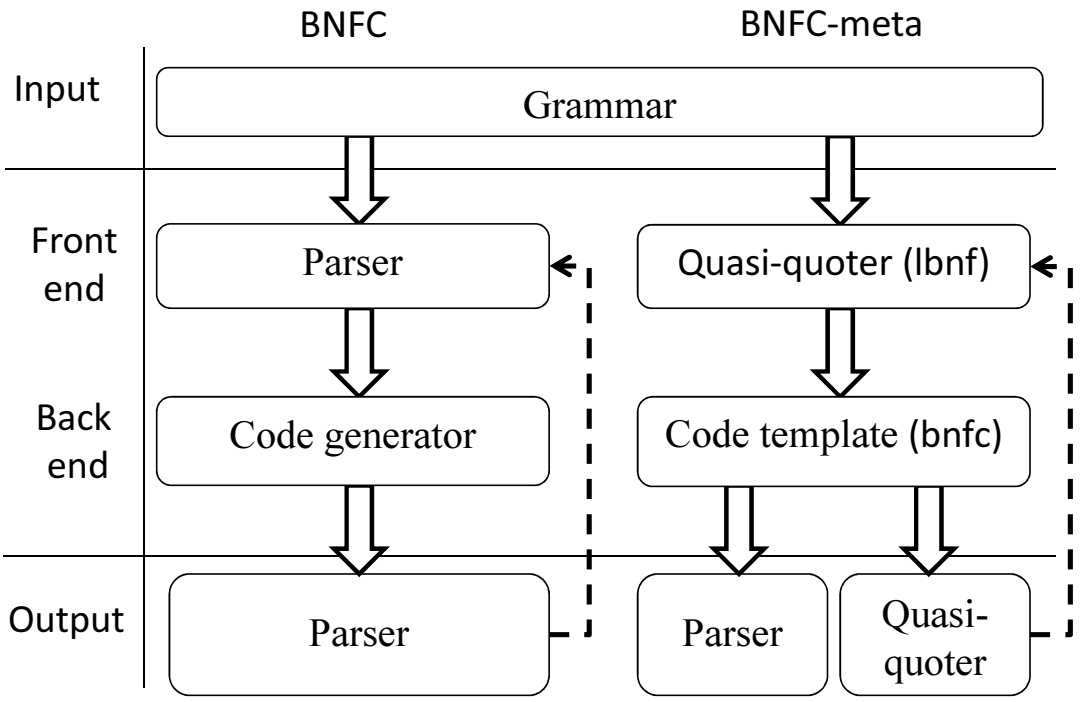

Figure 5. A comparative overview of BNFC and BNFC-meta

BNFC-meta example Consider our running example (the *Reg* language). We want to add these two modes of anti-quotation:

- Single identifiers, escaped by $, for example $\lambda[\, reg \mid \$a* \mid] \to a$ translates to $\lambda RStar\ a \to a$.
- General Haskell expressions or patterns, surrounded by angle brackets e.g. $f\ (r : rs) = [\, reg \mid \$r * <f\ rs> \mid]$ translates to $f\ (r : rs) = RSeq\ (RStar\ r)\ (f\ rs)$.

In order to implement the necessary syntactic changes to the grammar, we add two custom lexical tokens (as described in Section 2.1):

$$Aq1.\ Reg_3 ::= @\texttt{'<'}\ (char - \texttt{'>'}) * \texttt{'>'};$$
$$Aq2.\ Reg_3 ::= @\texttt{'\$'}\ letter\ (letter \mid digit)*;$$

The impact on the abstract syntax is as can be expected, two constructors ($Aq1$ and $Aq2$) are added to the *Reg* data type, each containing a string. This pollution of the abstract syntax types is detrimental to type safety: even though the constructors are intended to be used only for anti-quoting, the compiler cannot statically enforce that they are not used elsewhere.

To solve this problem, and to make defining anti-quotation syntax simpler in general, BNFC-meta introduces a special anti-quoting rule to the LBNF formalism. This rule automatically introduces anti-quotation as per the users request, without polluting the abstract syntax. In Figure 6 this rule is demonstrated. Syntactically AQ-rules are distinguished from regular rules by labels being replaced with a $ followed by an optional identifier. This instructs BNFC-meta quasi-quoters to parse any text that matches the rule as Haskell code. If a function name is written after the $, that function is used to parse the text (in our case we want to remove the enclosing angle brackets before passing the text to a Haskell parser for instance). In general the supplied function needs to be of type $Delta \to BNFC\_QQType$ where *Delta* is the AST type of the category on the right hand side of the rule. The type

```
module Regex where
import Language.LBNF
import RegexGrammars

bnfc $ combine extended [lbnf |
$ myAqE. Reg3 ::= @'<' (char - '>') * '>';
$ myAq.  Reg3 ::= @'$' letter (letter | digit)*;
|]

myAq :: String → BNFC_QQType
myAq = stringAq ∘ tail
myAqE :: String → BNFC_QQType
myAqE = stringAq ∘ reverse ∘ tail ∘ reverse ∘ tail
```

Figure 6. Adding anti-quotation to the regular expression language.

$BNFC\_QQType$ is more or less internal to the BNFC-meta library, the function *stringAq* can be used to parse a snippet of Haskell code into a $BNFC\_QQType$.

In Figure 7 we 1) define a general top-down traversal function on regexps and 2) use the function to transform regular expressions that don't use + and ? into equivalent ones that do. Note that the entire module is written without assuming anything about the abstract syntax other than the existence of a category named *Reg*. Thus, the names of constructors and other grammar details can be freely changed as long as the concrete syntax remains the same.

The BNFC-meta implementation Instead of producing a single parser and implementing quasi-quoters by translating the abstract syntax, BNFC-meta produces two parsers for each grammar category. One parser targets the abstract syntax, while the other targets

```
topdown :: (Reg → Reg) → Reg → Reg
topdown f rx = case f rx of
  [reg | $e1  $e2 |] → [reg | <r e1>    <r e2> |]
  [reg | $e1 | $e2 |] → [reg | <r e1>  |  <r e2> |]
  [reg | $e1 *    |] → [reg | <r e1> *       |]
  [reg | $e1 +    |] → [reg | <r e1> +       |]
  [reg | $e1 ?    |] → [reg | <r e1> ?       |]
  e                 → e
  where r = topdown f
transform :: Reg → Reg
transform = topdown step where
  step rx = case rx of
    [reg | $e | eps   |] → [reg | $e ?   |]
    [reg | $e1  $e2 * |]
      | e1 ≡ e2       → [reg | $e1 +  |]
      | otherwise     → rx
    e                 → e
```

Figure 7. Implementing a program transformation on regular expressions.

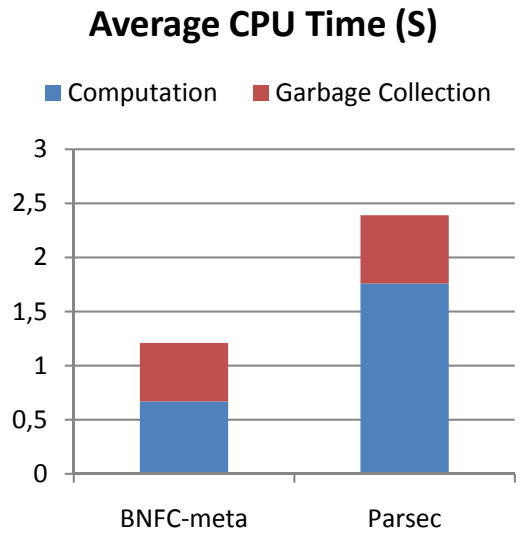

Figure 8. Average CPU usage when parsing a Mondrian file

Haskell expressions and patterns. This solution is not practical to write by hand, but that is not a problem for a metaprogram. The approach does not alter the abstract syntax and it guarantees that the original parser works and has no performance penalty in the presence of anti-quoting.

4.3 Error handling

Using metaprogramming in general, and quasi-quoters in particular, adds a new dimension to compile time error handling. There are no problems with errors "slipping through" to run time: all parsing is performed at compile time and the generated code is type checked after it is spliced. What can be problematic is the precision of the error messages, with regards to source location. The compiler will automatically indicate which quasi-quote the error occured in, anything beyond that will have to be provided by the programmer of the quasi-quoter. Syntax errors are reported well by BNF Converter parsers, with a source location and the concrete syntax leading up to the error. When syntax errors occur in quasi-quotes this information is presented to the user, in addition to the generic compiler error.

The situation is worse for type errors. With the basic quoters type errors can not occur unless there is a bug in BNFC-meta. In the presence of anti-quoting however, the user is free to introduce any imaginable type errors in the generated code. BNFC-meta has little control over the error messages in these situations, and they will be presented without an accurate source position and in terms of the generated code. This is a problem for quasi-quoters in general and not specific to BNFC-meta.

5. Performance and scalability

This section provides some examples and experimental results. All tests where performed on a 3.0 GHz Intel Xeon processor. Version 6.12.1 of the Glasgow Haskell Compiler was used, with optimisation level `-O2`.

5.1 Performance analysis

When using the quasi-quoters generated by BNFC-meta, performance is not a major issue (since the parsers are only used at compile time, when quasi-quotes are resolved). For applications that use the parsers at runtime there may be some requirements though. Since BNFC-meta is based on Alex / Happy the performance should be comparable to that of hand written parsers using them, which is usually considered good.

There are not many reliable performance comparisons between parser generators and parser combinators, possibly because it is difficult to make a fair comparison when the choice of object language and the amount of work put into each parser is very relevant to end performance. The general understanding seems to be that parser generators are faster. Parsec [Leijen and Meijer 2001] is considered a fast parser combinator library and it is seeing heavy use in the Haskell community.

For this reason we decided to compare the performance of Parsec and BNFC-meta. In order to avoid bias in the choice of object language or implementation of the Parsec parser we decided to use an existing Parsec parser and write an equivalent LBNF grammar for comparison. There are a number of examples included in the distribution of Parsec 2, some of them where excluded because they do not have context free syntax and as such can not be implemented in BNFC-meta. From the remaining, the largest example language was chosen and it proved to be a language called *Mondrian*. The source code for the Parsec parser (and abstract syntax) is a few hundred lines of Haskell code and the BNFC-meta implementation we wrote is around 50 lines of Haskell/LBNF code. The two implementations where considered "similar enough" to be comparable when the BNFC-meta parser could parse all the examples that the Parsec parser could, and the level of detail of the abstract syntax types seemed equivalent in a quick inspection. Figures 8 show the relative CPU time usage. The times are measured using the criterion benchmark tool. A very large Mondrian file (1 MB) was used in the test to ensure measurable time consumption. Initially we speculated that the time difference was caused by a difference in memory usage and specifically by garbage collection (GC), but this was not confirmed by measurements which showed only a slightly increased GC activity for Parsec.

Compilation time Since BNFC-meta does compile time calculations it is reasonable to assume that the compilation time should be somewhat greater than for Parsec. Also the amount of code that BNFC-meta produces can be quite big and the compiler needs to process and optimise it, which could increase compilation times further. This hypothesis is confirmed by our test results (see Fig-

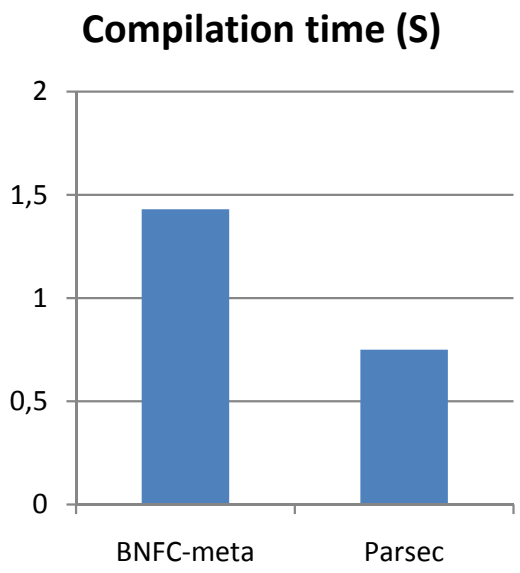

Figure 9. Compilation time of a Mondrian parser

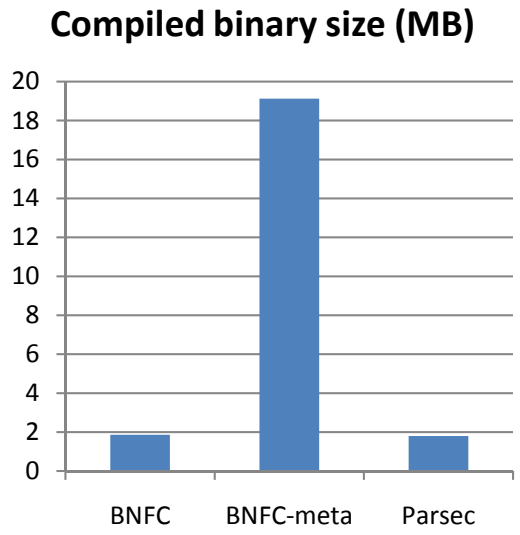

Figure 10. Size of produced binary for a Mondrian parser

ure 9), with the BNFC-meta parser taking twice as long to compile as the parsec one.

Binary size A somewhat troubling discovery was that the binary size of the BNFC-meta parser outweighs Parsec by an order of magnitude. Presumably this is because GHC fails to notice that some imported modules are only used at compile time, and thus some redundant components (e.g. a Haskell parser) are linked into the binary. As figure 10 shows, this hypothesis is supported by the fact that the original BNFC does not have this problem with the same grammar (although the BNFC and BNFC-meta parsers behave equally in all other tests).

5.2 Expressiveness

When comparing to flexible parser generators and combinator libraries such as parsec and hand written Alex/Happy it is important to note that BNFC-meta is limited to defining the same set of object languages as the original BNFC. This set is essentially all context-free languages and some context sensitive extensions like a limited kind of layout syntax (by automatic preprocessing). This means that it is difficult to capture many real world languages exactly using LBNF.

The fact that BNFC generates a lexer automatically and that this lexer does not distinguish newlines from white spaces imposes some further restrictions. Specifically it is not trivial to implement object languages with significant newline characters, which is a common feature. In principle it should be possible to give the user stronger control over the lexer without sacrificing the simplicity in cases where this is not needed. In spite of these limitations LBNF is no way limited to "toy languages", as we will demonstrate below.

5.3 A large scale example

In order to fit a complete definition, the running example in this article is very simple (only a single grammar category Reg). Nonetheless, the applications can scale up to fully equipped real world languages. To demonstrate this, we applied BNFC-meta to the grammar of (pre-processed) ANSI C available from the BNFC webpage. The grammar is roughly 250 LBNF rules so the module is too large to include in this paper, but not much larger than an average Haskell module. By default BNFC-meta generates quasi-quoters for any entry point categories (or for all categories if no entry point is specified in the grammar). In this case it means we are given quasi-quoters for C programs, statements and expressions. Using the $program$ quoter we can define a C program using concrete syntax:

```
p :: Program
p = [program |
int main () {
  printf ("Hello World");
  return 0;
}
 |]
```

The "least pretty equivalent" to this definitions (which only uses the abstract syntax constructors) looks like this:

```
p :: Program
p = Progr [Afunc (NewFunc [Type Tint] (NoPointer
  (OldFuncDec (Name (Ident "main")))) (ScompTwo
  [ExprS (SexprTwo (Efunkpar (Evar (Ident "printf"))
  [Estring "Hello World"])), JumpS (SjumpFive
  (Econst (Eoctal (Octal "0")))))])]
```

Anti-quoting There is an existing open source quasi-quoter library for C, called language-c-quote [Mainland 2009]. It offers a kind of "typed" anti-quotation where expressions like

$$[cexp \mid \$int : a + \$exp : b \mid]$$

means that a and b must be Haskell identifiers. Furthermore a must be an integer (i.e. of type Int) and b must be a C expression (i.e. of the same type as the quote).

We want to add a more liberal quotation syntax that allows more advanced Haskell expressions (not just single identifiers) in the quotes. We chose to have two anti-quotation modes for each category:

- An explicitly typed mode where $[C : h:]$ is a member of the category C for any Haskell expression h of type C (h may not contain the string `":]"` and may not contain nested quasi-quotes). Note the syntactic similarity with quasi-quoters.
- An implicitly typed mode where the category name is dropped meaning $[:h:]$ is a member of all categories.

```
import C
zeroFill :: Integer → [Ident] → Stm
zeroFill n arrs = [stm |
{
  int tmp;
  for (tmp = 0; tmp < [Integer : n:]; tmp++) {
    [:map zeroOneSlot arrs:]
  }
}
|]
zeroOneSlot :: Ident → Stm
zeroOneSlot arr = [stm |
  [Ident : arr:] [tmp] = 0;
|]
```

Figure 11. Using the C grammar with anti-quoting

Figure 11 demonstrates how these are used to define a program that is parameterised over an array length n and a list of identifiers (names of arrays), such that the produced C program fills each array with n zeroes. Note that the program deals only with the abstract syntax of C at runtime, so the Haskell type system statically guarantees that the program will produce syntactically correct code independent of the arguments to *zeroFill*. The second anti-quotation mode is controversial because it introduces massive ambiguities in the grammar. For instance if we have a quasi-quoted C statement $[stm \mid [:h:] \mid]$ then the anti-quote may refer to a statement, an expression or even an integer. The decision is made by the parser generator (automatically resolving ambiguities) but the user is warned of the ambiguities. In most cases assuming the wrong category will simply incur a type error, but if h is polymorphic there may be serious problems. One solution is to avoid the implicitly typed mode if the expression is polymorphic.

Observe that we describe the anti-quotation without mentioning that C is our object language. In fact this description of anti-quotation is expressed only in general terms of the existing grammar categories and as such it can be applied to any grammar. This has a tremendous potential for code reuse: instead of adding this mode of anti-quotation to the C grammar (and write a hundred or so grammar rules by hand) we can define a grammar transformation that adds this anti-quotation mode to any grammar. We can even parameterise the grammar transformation over the exact syntax of the anti-quotes; how quotes are initiated, what delimits the optional category name from the Haskell expression and how quotes are ended. The grammar transformation is thus a function:

```
antiquote ::
  String → String → String → Grammar → Grammar
```

And to add the desired anti-quotation to the C grammar we simply interpose this function between the quasi-quoter front end and the splicing back-end:

```
bnfc $ antiquote "[" ":" ":]" $ [lbnf |
... the C Grammar ...
|]
```

Implementing a general anti-quotation mode The implementation of *antiquote* adds two things to the grammar: a shared anti-quoting token and two anti-quotation rules for each category. The individual rules contain the start string `"["` and optionally the name of the category, the token matches the delimiter `":"` an arbitrary string and the ending string `":]"`. The LBNF rules for a single category C looks something like this:

```
$ g ::= "[C" AntiQuotingToken;
$ f ::= "[" AntiQuotingToken;
```

where *AntiQuotingToken* is the shared anti-quotation token. The functions g and f need to do the appropriate pruning of the matched string before the Haskell expression is parsed. Defining these rules is a simple task of collecting the categories of the original grammar, defining a new set of rules from these and adding this set to the original grammar.

While we would not go so far as to say that the generated quasi-quoters are equivalent to the ones provided by the language-c-quote library (such an analysis would not be practical), they are certainly quite similar. The most striking difference is the simplicity of the BNFC-meta implementation. Where language-c-quote has several hundred or even a few thousand lines of code, our implementation has a single Haskell source file with a readable and syntax-directed grammar definition. We also obtain a reusable pattern for anti-quoting and even for this single example the definition of this reusable pattern is shorter than the manual changes we would otherwise have needed.

6. Discussion

The closest predecessor to our work is the Quasi-quoting paper by Mainland [2007] where he gives the basic infrastructure to support user-defined quoting and anti-quoting in GHC. We build on, and extend this work in two directions. Firstly we demonstrate how to *generate* quasi-quoters (including flexible anti-quoting support) from labelled BNF. Secondly we give a method to embed other compiler-like tools as quasi-quoters.

We derive much of the strength of our tool from the implementation of the BNF Converter [Forsberg and Ranta 2003]. We internalise BNFC (+ Happy and Alex for parsing and lexing) and add general support for anti-quoting.

In his recent tutorial on combinator parsing Swierstra [2009] writes "Parser combinators occupy a unique place in the field of parsing: they make it possible to write expressions which look like grammars, but actually describe parsers for these grammars." In this paper we retain this property but also allow "off-line" parser generators to be used at compile time.

There is a long history of metaprogramming in the Lisp and Scheme community, starting already in the sixties. Twenty years ago *META* for Common Lisp used metaprogramming to construct parsers [Baker 1991]. The more recent survey of lexer and parser generators in Scheme [Owens et al. 2004] shows several examples and cites several Scheme-based generators.

From the C++ world, *Spirit* [de Guzman and Kaiser] is a set of libraries (part of the *Boost* library suite [Boost]) that can build recursive descent parsers, using an embedded grammar DSL. It uses template metaprogramming to generate parsers.

The comparison of DSL implementations by Czarnecki et al. [2003] looks at the meta-programming support of MetaOCaml, Template Haskell, and C++. All of those systems have the meta-programming strength to support compile time parser generation, but currently only Template Haskell supports quasi-quoting.

There are several attempts at bridging the gap between parser generators and combinators by improving combinator libraries. [Devriese and Piessens 2011] attempts to limit the problems related to left recursive grammars in parser combinator libraries. They use Template Haskell to perform some grammar transformation at compile time. In [Rendel and Ostermann 2010] a technique for getting "pretty-printers for free" from a parser combinator library is described.

6.1 Future work

While working on BNFC-meta we have found several interesting directions for future work. One direction is to work closely on the interface to and the implementation of Template Haskell. Here we would like to allow more control over error messages, especially when the following static checks on the generated code fail. We have also seen the need for generating not just declaration lists, but also module headers, including import and export lists. This would be helpful to avoid exporting some "junk" data types produced by Happy. Currently the only way to avoid polluting the name space of the module is to generate definitions in local **where** or **let** blocks, which is bad for error messages. Also we are not sure if the compiler responds well to a two thousand lines where clause.

Another direction we aim to pursue is to apply BNFC-meta to ongoing DSL projects (the Feldspar [Axelsson et al. 2010] backend, GPGPU programming with Obsidian [Svensson et al. 2010], language based security, etc.)

Finally we would also like to embed more libraries and tools in the same way and BNFC, Happy and Alex: candidates include the dependently typed language Agda and the Foreign Function Interface tool hsc2hs.

6.2 Conclusions

We use metaprogramming both to embed parser generators and to generate embedded parsers. There is a natural connection between these tasks: the latter provide a suitable front end for the first. This is evident from the bootstrapping ability of BNFC-meta.

The QuasiQuotes Haskell extension is a very general framework for quasi-quotation. Unfortunately this generality prevents such things as a general anti-quotation operator. By limiting ourselves to context free object languages and using quasi-quotation to produce abstract syntax, we can define these general modes of anti-quotation by transforming the object language definition on a grammar level.

As far as we know, BNFC-meta is the only system that provides both embedded parser generators and generation of embedded parsers in a single library.

References

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *MEMOCODE*, pages 169–178, 2010.

H. G. Baker. Pragmatic parsing in Common Lisp; or, putting defmacro on steroids. *SIGPLAN Lisp Pointers*, IV:3–15, April 1991. ISSN 1045-3563.

Boost. The Boost initiative for free peer-reviewed portable C++ source libraries. http://www.boost.org, 2009.

K. Czarnecki, T. O'Donnell, John, J. Striegnitz, and W. Taha. *DSL Implementation in MetaOCaml, Template Haskell, and C++*, volume 3016 of *LNCS*, pages 51–72. Springer-Verlag, Berlin, Heidelberg, 2003. doi: 10.1007/b98156.

J. de Guzman and H. Kaiser. Boost spirit. http://www.boost.org/doc/libs/1_46_1/libs/spirit/.

D. Devriese and F. Piessens. Explicitly recursive grammar combinators — a better model for shallow parser DSLs. In *Practical Aspects of Declarative Languages (PADL) 2011*, volume 6539 of *LNCS*. Springer, Jan. 2011.

M. Forsberg and A. Ranta. The BNF converter: A high-level tool for implementing well-behaved programming languages. In *NWPT'02 proc., Proc. Estonian Academy of Sciences*, December 2003.

D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Utrecht University, 2001. http://www.cs.uu.nl/~daan/parsec.html.

G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proc. ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5.

G. Mainland. The haskell package *language-c-quote*. http://hackage.haskell.org/package/language-c-quote, 2009.

R. C. Moore. Removing left recursion from context-free grammars. In *Proc. 1st North American chapter of the Association for Computational Linguistics conference*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

S. Owens, M. Flatt, O. Shivers, and B. Mcmullan. Lexer and parser generators in scheme. In *2004 Scheme Workshop*, 2004. URL http://repository.readscheme.org/ftp/papers/sw2004/owens.ps.gz.

A. Ranta. Grammatical framework. *J. Funct. Program.*, 14:145–189, March 2004. ISSN 0956-7968.

T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proc. third ACM Haskell symposium*, Haskell '10, New York, NY, USA, 2010. ACM.

T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop*, pages 1–16. ACM Press, 2002. ISBN 1-58113-605-6.

J. Svensson, K. Claessen, and M. Sheeran. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science*, 1(1):2065 – 2074, 2010. ISSN 1877-0509. doi: 10.1016/j.procs.2010.04.231. ICCS 2010.

S. D. Swierstra. *Combinator Parsing: A Short Tutorial*, pages 252–300. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6. doi: 10.1007/978-3-642-03153-3_6.