Why It's Nice to be Quoted: Quasiquoting for Haskell

Geoffrey B. Mainland
Harvard School of Engineering and Applied Sciences
mainland@eecs.harvard.edu

Abstract

Quasiquoting allows programmers to use domain specific syntax to construct program fragments. By providing concrete syntax for complex data types, programs become easier to read, easier to write, and easier to reason about and maintain. Haskell is an excellent host language for embedded domain specific languages, and quasiquoting ideally complements the language features that make Haskell perform so well in this area. Unfortunately, until now no Haskell compiler has provided support for quasiquoting. We present an implementation in GHC and demonstrate that by leveraging existing compiler capabilities, building a full quasiquoter requires little more work than writing a parser. Furthermore, we provide a compile-time guarantee that all quasiquoted data is type-correct.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Meta programming, quasiquoting

1. Introduction

Algebraic data types are one of most powerful hammers in the functional programmer's toolbox, allowing her to enforce invariants that aid reasoning about programs and catch errors at compile rather than run time. However, working with complex data types can impose a significant syntactic burden; extensive applications of nested data constructors are often required to build values of a given data type, or, worse yet, to pattern match against values. Anyone who has written a program that manipulates abstract syntax for a moderately complex language can appreciate the problem as well as the solution we propose: allow Haskell expressions and patterns to be constructed using domain specific, programmer-defined concrete syntax.

The Lisp world has long recognized the utility of automatically constructing program fragments via quasiquotation (Bawden 1999). Quasiquotation allows programmers to generate code automatically from code templates; the "quasi" in quasiquotation refers to the fact that these code templates can contain holes that are filled in by the programmer. The design of Scheme's hygienic macros (Kelsey et al. 1998) reflects decades of experience with quasiquoting and carefully considers many of the potential pitfalls surrounding quasiquotation, such as unintended variable capture. The Haskell world has Template Haskell (Sheard and Peyton Jones 2002) which similarly allows Haskell programs to construct other Haskell programs. These "program generating programs" are one type of metaprogram, programs that manipulate other programs as data. In both the Lisp and Template Haskell worlds, the language in which the metaprogram is written, or metalanguage, is identical to the language in which the manipulated programs are written, the object language. There are many cases when it would be useful to have an object language that is different from the metalanguage. The canonical example of a metaprogram is a compiler, which typically manipulates many different intermediate object languages before producing a binary. Other potential applications that could benefit from a more flexible quasiquoting system include peephole optimizers, partial evaluators, and any source-to-source transformation. The ability to quasiquote arbitrary object languages means the programmer can think about and write programs using the concrete syntax best suited to the domain, be it C, regular expressions, XML or some other language. Although we find support for quasiquoting arbitrary languages most compelling in the context of metaprogramming, quasiquoting is useful any time a complex data type can be given concrete syntax.

In this paper, we present an extension to the GHC Haskell compiler that allows expressions and patterns to be written using programmer-defined syntax extensions. Our contributions are:

- **A design for adding support for programmer-defined syntax extensions to GHC:** Our proposal builds on the work done to support Template Haskell (Sheard and Peyton Jones 2002), both syntactically and in its implementation. The syntactic scope of programmer-defined extensions in a source code file is clearly delimited, so programmers know exactly when they are writing user-defined syntax and when they are writing Haskell.
- **A scalable programming technique for writing quasiquoters:** Writing a syntax extension for our system requires only a small amount of effort beyond that already required to write a parser for the syntax in question. All additional effort is needed solely to support antiquotation. We show how to minimize this additional effort by leveraging the Scrap Your Boilerplate (SYB) (Lämmel and Peyton Jones 2003, 2004) approach to generic programming. Although we do not present the full details here, we have built a quasiquoter for ANSI C (with GCC extensions), so we know that our approach scales to real languages.
- **A working implementation of our design:** We have fully implemented our design as a patch against the current development version of GHC (6.7), consisting of slightly over 300 lines of code and available for download at the following URL: `http://www.eecs.harvard.edu/~mainland/ghc-quasiquoting/`.

Haskell'07, September 30, 2007, Freiburg, Germany.

The remainder of this paper is structured as follows. In Section 2 we motivate the benefit of quasiquoting through several examples involving different object languages. Using a small language, the untyped lambda calculus, we demonstrate in Section 3 that we can build and use a quasiquoter by doing little more than writing a parser for the object language being quoted. We discuss the type safety guarantees provided by quasiquoting—and the guarantees our approach cannot make—in Section 4. We say a few words about our implementation in GHC in Section 5 and explore related work in Section 6. In Section 7 we conclude and discuss possible future approaches to providing stronger static guarantees for quoted code.

2. Motivation

Haskell has a long history as a host language for embedding domain specific languages (Hudak 1998; Peterson et al. 1999; Leijen and Meijer 1999; Elliott et al. 2000; Pembeci et al. 2002). Typically these embedded languages make use of Haskell's type system by providing combinators that construct values representing terms in the embedded language. The usability of the combinator approach is enhanced by Haskell's support for declaring new infix operators. An alternative means to embedding a DSL is to provide a (necessarily partial) `compile` function that converts a string representation of a DSL term to its Haskell data type representation. Providing a `compile` function, as the Haskell `Text.Regex.Posix` library does (Kuklewicz), has the advantage of allowing language clients to write in a syntax that is not restricted to that offered directly by Haskell. The disadvantage is that terms in the embedded language cannot be checked until run time; voluntarily throwing away the advantages of a strongly typed language is a shameful act for all but the most minimal DSLs.

Flexible as Haskell's syntax may be, it is not a good fit for all language embeddings. Often the language designer reasons in terms of a concrete syntax but is forced to write in terms of combinators. Wouldn't it be better to directly support use of an arbitrary concrete syntax, thus freeing the programmer to think and write in the same language? Beyond the "mere" syntactic issue lies an issue of functionality: combinators may be just good enough for writing expressions in a DSL, but because patterns in Haskell are not first class, they can never be used to *match* against terms of a DSL. The string-based approach fails us here too.

Using our approach, the DSL designer provides a pair of functions that parse a DSL term's concrete syntax and return Haskell *abstract* syntax for the expression and pattern representation of the term, respectively. These parsers are run at compile time, so the resulting expressions (and patterns) are guaranteed to be type correct. The syntax we choose for quasiquotation is deliberately similar to the syntax used by Template Haskell for staged computations (Sheard and Peyton Jones 2002). Whereas Template Haskell quotes a Haskell expression using bracket-bar pairs, e.g., `[|1 + 2|]`, we specify the concrete syntax being quoted with an additional colon and identifier following the initial open bracket. The identifier must be bound to a Haskell tuple whose constituent members are parsing functions for expressions and patterns, respectively. Figure 2 shows a C function that is quasiquoted using our C quasiquoting library. In the scope in which the quasiquotation occurs, `cfun` is bound to a tuple containing parsers that take a string as input and produce Haskell abstract syntax for the corresponding expression and pattern, respectively. The expression $int : n$ is an *antiquotation* and causes, at run-time, the value of n to be spliced into the abstract syntax tree for the quasiquoted C function `add`.

DSL designers strive to ensure that the syntactic correctness of their language is guaranteed at compile time. Through proper staging, quasiquoters make this goal easier to achieve—there is potential for more work to be done at compile time, including compile-time optimizations of DSL terms, because DSL terms can be fully constructed by a quasiquoter at compile time instead of by combinators at run time. We now turn to several examples of languages embedded in Haskell and demonstrate how support for quasiquoting makes the jobs of the embedded language's designer and its users easier.

```
add n =
  Func (DeclSpec [] [] (Tint Nothing))
       (Id "add")
       DeclRoot
       (Args
           [Arg (Just (Id "x"))
                (DeclSpec [] [] (Tint Nothing))
                DeclRoot]
           False)
        (Block []
          [Return (Just
            (BinOp Add
                   (Var (Id "x"))
                   (Const (IntConst (False, n))))))])
```

Figure 1: **Haskell syntax for the `add` function in the absence of quasiquotation.**

2.1 Quasiquoting C

Writing a compiler usually involves the use of several intermediate languages. GHC itself has used many intermediate languages over the years, including, but not limited to, GHC Core (Tolmach), "Abstract C" (Peyton Jones 1992) and C-- (Peyton Jones et al. 1999). Sometimes these languages have a true external form, but they almost always have an external form that at least exists as a convention used by the developers when reasoning and discussing the internal workings of the compiler. Providing concrete syntax for these languages allows the programmer to write as she thinks; translating ideas from the blackboard to implementation is direct, and reasoning about code written in concrete syntax is easier.

Our own experience in embedding domain specific languages led to the present work on quasiquotation. The embedded language Flask (Mainland et al. 2006) is a dataflow language for sensor networks that describes computations over streams of data. Programmers construct "dataflow graphs" that are then compiled to NesC (Gay et al. 2003) and run on sensor network devices that have 16 bit CPUs and 10K of RAM. Operators in the dataflow graph, such as `map`, are parametrized by NesC code, so it is vital that programmers be able to directly write NesC code when constructing dataflow graphs. Figure 2 shows a Haskell function, written using our quasiquoting library, that takes a Haskell `Integer` and returns abstract syntax for a C function that adds that integer to its argument and returns the result. The same function written directly in Haskell without any syntax extensions is shown in Figure 1. Although the direct Haskell version is (barely) readable, it is not tolerably writable. A library of combinators would certainly ease this pain, but direct support for C's syntax is the ideal solution. For even small C functions the benefit of using concrete syntax is already apparent. This payoff for allowing the direct use of C concrete syntax is even greater in the context of Flask, where programmers often write large chunks of NesC code. Even a library of combinators is a significant syntactic burden in these circumstances.

The readability problem becomes even more acute when instead of constructing values we wish to *deconstruct* them via pattern matching. Combinators are no help to us because patterns in Haskell are not first class; the ability to quasiquote patterns makes programs much more readable. Figure 3 shows a function that per-

```
add n = [:cfun |
  int add (int x)
  {
    return x + $int : n$;
  }
|]
```

Figure 2: **Haskell syntax for the `add` function in the presence of quasiquotation support.**

```
cfold :: Data a
      ⇒ a
      → a
cfold a = everywhere (mkT f) a
  where
    f :: Exp → Exp
    f [:cexp | $int:x$ + $int:y$|] = [:cexp | $int:x + y$|]
    f [:cexp | $int:x$ - $int:y$|] = [:cexp | $int:x - y$|]
    f [:cexp | $int:x$ * $int:y$|] = [:cexp | $int:x * y$|]
    f [:cexp | $int:x$ / $int:y$|] = [:cexp | $int:x/y$|]
    f [:cexp | $int:x$ % $int:y$|] = [:cexp | $int:x `mod` y$|]
    f exp                         = exp
```

Figure 3: **Implementing constant folding for C.**

forms bottom-up constant folding on a C parse tree, making use of the SYB (Lämmel and Peyton Jones 2003) `Data` type class and functions `everywhere` and `mkT` to apply the function `f` to the parse tree using a bottom-up traversal. Although we would expect this "optimization" to be of little use when our parse tree is destined for compilation by a production C compiler, for intermediate languages used by a compiler this sort of optimization is typical. Instead of being obfuscated by a mess of nested constructor applications, the constant folding transformation's effects on an abstract syntax tree are immediately clear to any reader.

The Pan embedded DSL for image creation and manipulation (Elliott et al. 2000) also generates C code from DSL expressions. Unlike Pan, Flask terms are actually parametrized by C code, so our need for quasiquoting is more pressing. However, Pan's code generation facilities would undoubtedly benefit from the use of our C quasiquoting library.

2.2 An x86 Peephole Optimizer

Peephole optimizers operate on streams of assembly instructions, typically replacing short instruction sequences by more efficient sequences. Pattern matching is of fundamental importance to peephole optimization, and a clear demonstration of the advantage of pattern quasiquotation. Figure 4 shows one case of a peephole optimizer in which a redundant comparison instruction is eliminated. Here antiquoted values are signified with a leading `&` character since the `$` character is already part of the assembler's syntax. The pattern shown in the code binds the variables `s`, `r1`, `r2`, `r3`, `r4` and `lbl`; these bound variables are then used to produce the optimized assembly instruction sequence.

The advantage to representing peephole optimization—or in general any data transformation—using concrete syntax is that the transformation becomes much easier to reason about and maintain. Code written in this style is self-documenting, whereas the comparison elimination written using the standard data constructor application syntax would require a separate description of what was actually going on to be easily understood.

```
peep :: [Asm] → [Asm]
peep [:asm | mov&s $&r1, &r2
             cmp   $&r3, &r4
             je    &lbl |] : rest
      | r3 ≡ r1 ∧ r4 ≡ r2
  = [:asm | mov&s $&r1, &r2
             jmp   &lbl |] : rest
...
```

Figure 4: **One case of a simple x86 peephole optimizer.**

2.3 Regular Expressions

As previously mentioned, the Haskell `Text.Regex.Posix` regular expression library parses regular expressions at run time, rather than compile time. Consider the following code to construct a regular expression object:

```
let r = mkRegex "(foo"
```

Despite being obviously wrong, this code fragment will compile. The programmer's error will manifest itself as a runtime error. With quasiquoting support we can do better:

```
let r = [:re | (foo |]
```

Now our error becomes a compile-time error because the quasiquoter `re` runs *when the program is compiled* rather than when the program is executed.

2.4 XML

There is a small industry in the functional languages community geared towards domain specific languages for manipulating and processing XML, from HaXML (Wallace and Runciman 1999), which is embedded in Haskell, onward (Atanassow et al. 2003; Hosoya et al. 2005; Hosoya and Pierce 2003; Benzaken et al. 2003). Embedded languages like HaXML are open to two possible representation alternatives for XML documents, an internal data structure that can represent the contents of any XML document that nonetheless provides structure beyond simple strings, and a representation based on Haskell data types derived from a DTD definition. The former offers flexibility, and the latter offers the static guarantee that only well-formed XML output will be generated. One of the key benefits of quasiquoting, particularly in the context of XML processing, is that it allows the programmer to write code that is neutral to the choice of representation for the code being quoted. By allowing XML fragments to be written in concrete XML syntax, the program text is decoupled from the representation of XML documents, and the programmer is free to move between typed and untyped representations without having to rewrite code.

Figure 5 shows a DTD for books (taken, slightly modified, from (Atanassow et al. 2003)) and its HaXML translation to Haskell data types. Although elided here, this translation also includes a parser and printer for the translated data types; the parser can easily be extended to allow inlining of XML in Haskell code using the technique described in Section 3. Assuming this quoter is bound to the variable `book`, the following is legal code in the presence of our extension:

```
soe = [:book|
  <book>
    <title> The Haskell School of Expression </title>
    <author> Paul Hudak </author>
    <date>    2000 </date>
    <chapter> Problem Solving, Programming,
              and Calculation
```

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
```

```
data Book = Book Title Author Date [(Chapter)]
  deriving (Eq, Show)
newtype Title = Title String
  deriving (Eq, Show)
newtype Author = Author String
  deriving (Eq, Show)
newtype Date = Date String
  deriving (Eq, Show)
newtype Chapter = Chapter String
  deriving (Eq, Show)
```

Figure 5: **The book DTD and its corresponding translation to Haskell data types.**

```
    </chapter>
    <chapter> A Module of Shapes: Part I </chapter>
    <chapter> Simple Graphics </chapter>
  </book>
|]
```

Switching between the typeful representation of XML and the typed representation only requires redefining the value of `book`, likely a one line change. By using concrete syntax, the programmer is insulated from this change in representation.
Thanks to our regular expression quasiquoter, we can now search for books by Paul Hudak:

```
hudakSearch :: [Book] → [Book]
hudakSearch books = filter f books
  where
    f :: Book → Bool
    f (Book _ (Author auth) _ _)
      | Just _ ← [:re | Hudak |] `matchRegex` auth
          = True
    f _   = False
```

3. The Gritty Details: Writing a Quasiquoter

At least in the case of expressions, quasiquoting can be done using only Template Haskell by splicing a Haskell expression that calls a parsing function with a string as its argument, and indeed that is almost exactly how we implement expression quasiquoting in GHC. There are three contributions of our technique that go beyond what pure Template Haskell has to offer, two of which are of technical merit and one of which is important from a usability standpoint. First, unlike Template Haskell, our quasiquoting system allows *patterns*, including binding occurrences of pattern variables, to be quoted—support for splicing patterns is currently missing from GHC. Second, we show how to use the SYB approach to generic programming to reflect values back into the language, which greatly facilitates writing quasiquoters and is useful even in the pure Template Haskell world. The third contribution we make is that quasiquotation parsers are handed a source code location in addition to the string to be parsed. This is of vital importance in terms of usability as it allows syntax errors within quoted code to be pinpointed precisely.

We illustrate the design of our quasiquoting system through a simple quoted language: the untyped lambda calculus. Our implementation progresses from a standard parser and evaluator to a full quasiquoter with support for antiquotation. Although this example is simple, it touches on all aspects of our quasiquoting system including both the use of a quasiquoter and the details of its implementation. More complex object languages will of course require more work, but in our experience building a full quasiquoter for C, we found that the techniques we describe in this section scale.

Our simple untyped lambda calculus implementation is shown in Figure 6, and the parser's definition is shown in Figure 7. Obviously we have yet to make use of quasiquoting. As a first step, let us consider the case of `subst` that handles application. Instead of using abstract syntax, we would like to use concrete syntax to specify both the pattern binding `e1` and `e2` and the expression that returns the application of `e1'` to `e2'`. The new application case for substitution is:

```
subst [:lam | $exp:e1 $exp:e2 |] x y =
  let e1' = subst e1 x y
      e2' = subst e2 x y
  in
    [:lam | $exp:e1' $exp:e2' |]
```

As previously mentioned, the syntax we choose for quasiquotation is deliberately similar to the syntax used by Template Haskell for staged computations (Sheard and Peyton Jones 2002). In our rewritten `subst` function, `lam` is bound to a pair of parsers, `lame` and `lamp`. The function `lame` returns abstract syntax for a Haskell expression, and the `lamp` function returns abstract syntax for a Haskell pattern.

The new case for application also makes use of antiquotation: the four variables `e1`, `e2`, `e1'` and `e2'` are all Haskell variables, not lambda calculus variables. The dollar sign indicates antiquotation, and `exp:` indicates that an expression is being antiquoted. For our small lambda language there are two syntactic categories we wish to antiquote: variables and expressions. This syntax is specific to the particular language being quoted, and in general there will be more than two syntactic categories the programmer will want to antiquote.

Because `lame` and `lamp` return abstract syntax for Haskell patterns and expressions, respectively, we need a data type that represents Haskell abstract syntax. Fortunately Template Haskell provides a convenient library containing just the data types we need as well as functions for manipulating these data types. The quotation parsers `lamp` and `lame` make use of this library, and have the types[1]:

```
lame :: (String, Int, Int) → String → TH.ExpQ
lamp :: (String, Int, Int) → String → TH.PatQ
lam = (lame, lamp)
```

The first argument is the source code location of the start of the string being parsed, consisting of a file name, line number and column. The second argument is the text to be parsed. The result is a value in Template Haskell's *quotation monad*.

3.1 Maximizing Parser Re-use

We now have three functions that all must parse the concrete syntax for lambda expressions: our original parser, the parser that produces Haskell abstract syntax for expressions, and the parser that produces Haskell abstract for patterns. Two of these parsers, those for Haskell expressions and patterns, must also handle anti-quotation. We would like to re-use as much of our parser `parse` as possible.

[1] Throughout our examples we use the qualified package name `TH` as an abbreviation for `Language.Haskell.TH`

```
data Var = V String
  deriving (Eq)

data Exp = Var Var
         | Lam Var Exp
         | App Exp Exp

allBinders :: [Var]
allBinders = [V [x]           | x ← ['a'..'z']] ++
             [V (x : show i) | x ← ['a'..'z'],
                               i ← [1 :: Integer ..]]

free :: Exp → [Var]
free (Var v)     = [v]
free (Lam v e)   = free e \\ [v]
free (App e1 e2) = free e1 `union` free e2

occurs :: Exp → [Var]
occurs (Var v)     = [v]
occurs (Lam v e)   = v : occurs e
occurs (App e1 e2) = occurs e1 `union` occurs e2

subst :: Exp → Var → Exp → Exp
subst e x y = subst' (allBinders \\ occurs e `union` occurs y) e x y
  where
    subst' :: [Var] → Exp → Var → Exp → Exp
    subst' _     e@(Var v)      x y
        | v ≡ x     = y
        | otherwise = e
    subst' fresh e@(Lam v body) x y
        | v ≡ x           =      e
        | v ∈ free y      =      Lam v' (subst' fresh' body' x y)
        | otherwise       =      Lam v (subst' fresh body x y)
      where
        v' :: Var
        fresh' :: [Var]
        (v' : fresh') = fresh
        body' :: Exp
        body' = subst' (error "fresh variables not so fresh")
                       body v (Var v')
    subst' fresh (App e1 e2)      x y =
        let e1' = subst' fresh e1 x y
            e2' = subst' fresh e2 x y
        in
          App e1' e2'

eval :: Exp → Exp
eval e@(Var _)   = e
eval e@(Lam _ _) = e
eval (App e1 e2) =
  case eval e1 of
    Lam v body  → eval (subst body v e2)
    e1'         → App e1' (eval e2)
```

Figure 6: **Abstract syntax and evaluator for the untyped lambda calculus.**

Ignoring the problem of antiquotation for a moment, there are two possible solutions:

1. Write one-off functions that convert values with types `Var` and `Exp` to an appropriate Haskell abstract syntax representation. Doing so would require four functions in our case and is tedious and error-prone even for the small lambda language example.
2. Copy and paste, creating two new versions of the parser. One version will directly return Haskell abstract syntax for a Haskell pattern, and the other will return Haskell abstract syntax for an expression. This is potentially a maintenance nightmare. Furthermore, we lose a lot of the benefits of the type checker: a value of type `TH.ExpQ` is Haskell abstract syntax for an expression, but knowing this tells us nothing about the type of the Haskell expression represented by the abstract syntax. This expression could be an `Integer`, a `String` or have any other type—we know only that it is *syntactically* correct, not that it is *type* correct.

```
parens p = between (symbol "(") (symbol ")") p

whiteSpace = many $ oneOf " \t"

small = lower <|> char '_'
large = upper
idchar = small <|> large <|> digit <|> char '\''

lexeme p = do x ← p
              whiteSpace
              return x

symbol name = lexeme $ string name

ident :: CharParser () String
ident = lexeme $
        do c ← small
           cs ← many idchar
           return $ c : cs

var :: CharParser () Var
var = do v ← ident
         return $ V v

exp ::  CharParser () Exp
exp = do es ← many1 aexp
         return $ foldl1 App es

aexp :: CharParser () Exp
aexp = (try $ do v ← var
                 return $ Var v)
   <|> do symbol "\\"
          v ← var
          symbol "."
          e ← exp
          return $ Lam v e
   <|> parens exp

parse :: Monad m ⇒ String → m Exp
parse s =
    case runParser p () "" s of
      Left err → fail $ show err
      Right e  → return e
  where
    p = do e ← exp
           eof
           return e
```

Figure 7: **Parser for the untyped lambda calculus.**

Option 1 would be much more appealing if we could write *generic* functions that convert a value of any type into Haskell abstract syntax representing that value. Then we could simply compose `parse` with such a generic function and the result would be a quasiquoter. As it turns out, this is quite easy to do using the SYB approach to generic programming, support for which is included in GHC (Lämmel and Peyton Jones 2003, 2004). The astute reader will note that the `parse` function does not handle antiquotation. Using generic programming we can in fact accommodate antiquo-

tation, but to simplify our explanation we will temporarily ignore this detail.

To use the SYB approach to generic program we must slightly modify the `Var` and `Exp` data types and add `deriving` clauses so that instances for the `Data` and `Typeable` classes are automatically generated by GHC. Adding these automatic derivations reflects information about the data types into the language so that we can now manipulate values of these types generically. We need two generic functions: one that converts a value to Haskell abstract syntax for a pattern representing that value, and one that converts a value into Haskell abstract syntax for an expression representing the value. The functions `dataToExpQ` and `dataToPatQ`, defined in the Appendix, are just the functions we desire. With these two simple functions, any value of a type that is a member of the `Data` type class can be converted to its representation in Haskell abstract syntax as either a pattern or an expression. This allows us to trivially write `lame` and `lamp` as follows:

```
lame :: (String, Int, Int) → String → TH.ExpQ
lame _ s = parse s >>= dataToExpQ

lamp :: (String, Int, Int) → String → TH.PatQ
lamp _ s = parse s >>= dataToPatQ
```

By using generic programming, we can take a parser and create expression and pattern quasiquoters for the language it parses with only *four* lines of code, including type signatures! This holds not just for our simple object language, but for any object language.

3.2 Adding Support for Antiquotation

Without antiquotation our quasiquoters are not very useful—they can only be used to write constant patterns and expressions. Adding support for antiquotation is a must to make quasiquoting useful and can be done with only slightly more than four lines of code. First we must extend our abstract syntax to include support for antiquotes. Changing the parser is unavoidable, but we can still write a single parser and reuse it to parse pattern quasiquotes, expression quasiquotes and plain syntax without any antiquotation by setting an appropriate flag in the parsing monad. The key point here is that in all three case the parser is producing a value with the type of whatever data type is used to represent the object language's abstract syntax.

SYB defines combinators that extend a generic function with type-specific cases. We use these combinators to convert antiquotes in the object language to appropriate Haskell abstract syntax. Figure 8 shows all code required to support full quasiquotation for the lambda language, not including changes to the parser which are shown in Figure 9. The two new data constructors `AV` and `AE` are for antiquoted variables and expressions, respectively. For each syntactic category that is antiquoted, two additional functions must be written: one to generate the appropriate Haskell abstract syntax for patterns, and one to generate Haskell abstract syntax for expressions. These functions are combined using the `extQ` SYB combinator to form a single generic function, and this function is then passed to the function that reifies values as Haskell abstract syntax (either `dataToExpQ` or `dataToPatQ`).

Although this technique minimizes the changes one must make to a parser to add support for antiquotation, it has the unfortunate requirement that we must also modify the data type used by the parser. Ideally we could *extend* the original data type used to represent abstract syntax to add support for antiquotation constructs; this is an instance of the expression problem, formulated by Wadler (Wadler 1998). A recent proposal for solving the expression problem in Haskell by providing direct support for open data types and open functions (L oh and Hinze 2006) would benefit quasiquoters everywhere, but our approach is nonetheless minimally intrusive.

```
data Var = V String
         | AV String
  deriving (Eq, Typeable, Data)

data Exp = Var Var
         | Lam Var Exp
         | App Exp Exp
         | AE String
  deriving (Typeable, Data)

antiVarE :: Var → Maybe TH.ExpQ
antiVarE (AV v) = Just $ TH.varE $ TH.mkName v
antiVarE _      = Nothing

antiExpE :: Exp → Maybe TH.ExpQ
antiExpE (AE v) = Just $ TH.varE $ TH.mkName v
antiExpE _      = Nothing

antiVarP :: Var → Maybe TH.PatQ
antiVarP (AV v) = Just $ TH.varP $ TH.mkName v
antiVarP _      = Nothing

antiExpP :: Exp → Maybe TH.PatQ
antiExpP (AE v) = Just $ TH.varP $ TH.mkName v
antiExpP _      = Nothing

lame :: (String, Int, Int) → String → TH.ExpQ
lame _ s = parse s >>=
  dataToExpQ (const Nothing `extQ` antiVarE
                            `extQ` antiExpE)

lamp :: (String, Int, Int) → String → TH.PatQ
lamp _ s = parse s >>=
  dataToPatQ (const Nothing `extQ` antiVarP
                            `extQ` antiExpP)
```

Figure 8: **Code required to support full quasiquotation for the lambda language (not including changes to the parser).**

```
var :: CharParser () Var
var =  ...
  <|> do string "$var:"
         v ← ident
         return $ AV v

aexp :: CharParser () Exp
aexp = ...
  <|> do string "$exp:"
         v ← ident
         return $ AE v
```

Figure 9: **Changes to the untyped lambda calculus parser required to support antiquotation.**

It should also be noted that the approach we have outlined here only generates Haskell abstract syntax for constructor applications—the output of a quasiquotation will never be a lambda term. Of course quasiquoters are free to generate any Haskell abstract syntax they wish, including lambda terms, but this will require more work on the part of the quasiquoter writer. It will also complicate the reuse of an existing parser that directly generates abstract syntax values. In other words, for object languages that are represented using an abstract syntax data type, parser re-use comes almost for free; for object languages that must in general be "compiled" to Haskell terms with sub-terms that are lambda expressions there is extra work to be done.

4. Type Safety Guarantees

All quasiquoters are run at compile time, so any parsing errors or errors in generated Haskell abstract syntax will therefore be caught at compile time. Furthermore, all generated Haskell abstract syntax must pass the type checker. We can state the safety guarantee that holds for compiled quasiquoted code as follows: any invariant that holds for the data type that represents the abstract syntax for the quasiquoted code also holds in the compiled program. If we were to use quasiquotation to construct large expressions in our lambda language and output them as text, this safety guarantee would statically ensure that all output lambda expressions were syntactically correct. For the more sophisticated C quasiquotation system, our safety guarantee *statically* ensures that all generated C code is syntactically correct (assuming that any value whose type is that of C abstract syntax can be printed as valid concrete C syntax).

However, our quasiquoter for the C language cannot statically guarantee that any generated C code is type correct with respect to C's type system unless this invariant can somehow be encoded in the abstract syntax representation used by the quasiquoter. One could imagine that the C parser could also perform type checking, but this would still not resolve the issue in the presence of antiquotation because of the open code problem. Consider the following quasiquoted C code:

```
int inc($ty:t$ x)
{
    return x + 1;
}
```

Here we have antiquoted the type, `t`, of the argument to the function `inc`. A C parser cannot type check this code because it cannot know what type `t` represents! In general we cannot make any static guarantees about the type-correctness of generated C code—we can only guarantee that it is syntactically correct. Using GADTs (Xi et al. 2003) allows a static type safety guarantee to be enforced for some quoted languages. In general if the object language's type system can be embedded in Haskell's type system, then using an appropriate GADT encoding we can statically guarantee that all quasiquoted code is type correct with respect to the object language's type system. We leave a more thorough exploration of this question to future work.

5. Implementation

Our implementation of quasiquoting in GHC is in the form of a patch against GHC 6.7 consisting of about 300 lines of code. We reused much of the machinery that already exists in GHC to support Template Haskell. Supporting quasiquoting of expressions was a trivial addition because GHC already supports quoting of Haskell expressions—we only had to add code to call the quasiquoter. Regrettably, GHC does not support Template Haskell's pattern quotation facility at all and generates a compile-time error if pattern quotations are used. Adding full support for Template Haskell's pattern quotation was a larger chunk of work than we were willing to bite off, so we limited ourselves to supporting only the pattern quotation mechanism described in this paper. This necessitated a fair amount of additional work to handle the binding occurrences that arise from antiquotation of patterns.

6. Related Work

A great deal of work has been done on metaprogramming in the functional language community, including MetaML (Taha and Sheard 1997), MetaOCaml (Taha 2003) and Template Haskell (Sheard and Peyton Jones 2002). In these systems the object language (the quoted language) is always the same as the metalanguage. MetaML and MetaOCaml provide additional type checking for quoted code; in MetaOCaml the quoted expression `.<1+2>.` has type `int code` instead of just type `code`. Template Haskell assigns all quoted code the same type. While we agree with the authors of these systems that metaprogramming is an important tool, we believe that it is equally important to provide access to many object languages by allowing for extensible quasiquoting. Allowing the metaprogrammer to manipulate programs in any language she chooses instead of restricting her to work exclusively with the same language at both the meta- and object level greatly expands the possible applications of metaprogramming.

The system that bears the most similarities to our work is camlp4 (de Rauglaudre 2003). In fact we were motivated to add support for quasiquoting to GHC after using camlp4 in a substantial metaprogramming application. Unlike our system, one of the goals of camlp4 is to allow the programmer to arbitrarily change the syntax of the host language. We wish only to add support for providing concrete syntax for data. Quasiquotaton modules also run at compile time in camlp4, so they provide the same static safety guarantee that our system provides. However, we believe that Haskell's type system, in particular GADTs, will allow stronger invariants to be encoded in data types so that more than syntactic correctness of generated code can be statically verified. The major advantage of our approach over that of camlp4 is that we demonstrate how to use generic programming to reuse a single parser to parse quasiquoted patterns, quasiquoted expressions and plain syntax that does not include antiquotes. Because OCaml does not support generic programming out of the box, in camlp4 this would require three separate parsers, each generating different representations of the same concrete syntax.

Baars and Swierstra's work on syntax macros (Baars and Swierstra 2004) aims to provide functionality similar to camlp4 in the context of Haskell. Although more general than our approach, syntax macros are unfortunately not available in GHC. We aim to make a small, conservative extension to existing GHC functionality narrowly-focused on supporting programmer-defined concrete syntax for complex data types, not to provide a general-purpose mechanism for redefining the language accepted by the compiler. Baars and Swierstra also use phantom types and explicit evidence passing to enforce invariants on typed abstract syntax that go beyond mere syntactic correctness, although GADTs now provide the same functionality (and then some) with less effort.

Wadler's proposal for views (Wadler 1987) allows pattern matching to be abstracted away from the data type being matched. Our work is orthogonal to the work on views: our goal is to provide a mechanism for describing patterns in terms of programmer-defined concrete syntax. Closer to our work is the work on first class patterns (Tullsen 2000). First class patterns would allow embedded DSL designers to define combinators for pattern matching as well as term generation, but we still believe that even in the presence of first class patterns quasiquoting is a desirable feature. In any case, neither views nor first-class patterns are implemented in any real-world Haskell compiler; quasiquotation is implemented and available today.

7. Conclusions and Future Work

Quasiquoting is a powerful tool. By making programs easier to read and write through providing concrete syntax for describing data, it also aids the programmer in reasoning about her programs. Because quasiquoting operations are all performed at compile time, any invariant that is enforced by a data type is statically guaranteed to hold for quasiquoted data of that type. These benefits are not only significant, but cheap. By leveraging generic programming, writing a full quasiquoter requires little more work than writing a parser.

We expect that many Haskell programmers will immediately put this new tool to use.

It remains to be seen how best to address the typing issues that arise when using quasiquoting. It should be noted that these issues are not new, but arise in any metaprogramming system. They are simply more apparent in our system because we support an unlimited number of object languages and have already addressed the low-hanging fruit by providing a static guarantee that generated code is syntactically correct. We alluded to one problem with open code in Section 4. Another type of open code is that in which the code has free variables at the object language level rather than free variables at the metalanguage level introduced by antiquotation. For example, consider the MetaOCaml quoted code `.<x>.` where the variable `x` is free. What type should we assign this code fragment?

This open code problem is not easily solved. MetaML and MetaOCaml allow free variables in quoted code as long as they are lexically bound in the surrounding metalanguage. This solution would not necessarily work when the object language and metalanguage are not the same. It is also somewhat unsatisfying—we may wish to allow free variables in open code that are lexically bound by a context into which the quoted code is later spliced. If we were to frame the type checking problem as a constraint problem, then open code could carry a set of type constraints that would be statically checked against all possible contexts in which the quoted code could be spliced. Allowing each object language to provide its own constraint generating and constraint solving engines could allow us to guarantee not only that all generated code is syntactically correct, but also that it is type correct. We leave the exploration of such an extensible type system to future work.

References

F. Atanassow, D. Clarke, and J. Jeuring. Scripting XML with generic haskell. Technical Report UU-CS-2003, Institute of Information and Computing Sciences, Utrecht University, 2003.

Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017485.

Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999. URL `citeseer.ist.psu.edu/bawden99quasiquotation.html`.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9): 51–63, September 2003.

Daniel de Rauglaudre. Camlp4 reference manual, 2003. URL `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`.

Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG*, pages 9–27, 2000. URL `citeseer.ist.psu.edu/elliott00compiling.html`.

David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/767193.767195.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, January 2005. Preliminary version in ICFP 2000.

Paul Hudak. Modular domain specific languages and tools. In *ICSR 98*, 1998. URL `http://haskell.org/frp/dsl.pdf`.

S. Kamin. Standard ML as a meta-programming language. Technical report, University of Illinois at Urbana-Champaign, 1996. URL `citeseer.ist.psu.edu/kamin96standard.html`.

Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33 (9):26–76, 1998. URL `citeseer.ist.psu.edu/kelsey98revised.html`.

Chris Kuklewicz. `Text.Regex.Posix`. `http://haskell.org/ghc/docs/6.6.1/html/libraries/regex-posix/Text-Regex-Posix.html`.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3): 26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proc. 2nd USENIX Conference on Domain-Specific Languages'99*, 1999. URL `http://research.microsoft.com/~emeijer/Papers/HaskellDB.pdf`.

Andres L oh and Ralf Hinze. Open data types and open functions. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 133–144, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-388-3. doi: http://doi.acm.org/10.1145/1140335.1140352.

Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A language for data-driven sensor network programs. Technical Report TR-13-06, Harvard University, 2006.

Izzet Pembeci, Henrik Nilsson, and Greogory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Principles and Practice of Declarative Programming (PPDP'02)*, October 2002. URL `http://haskell.cs.yale.edu/yale/papers/ppdp02/ppdp02.ps.gz`.

John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551: 91–105, 1999. URL `citeseer.ist.psu.edu/peterson99lambda.html`.

Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999. URL `http://www.eecs.harvard.edu/~nr/pubs/c--gc-abstract.html`.

Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992. URL `citeseer.ist.psu.edu/peytonjones92implementing.html`.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

Andrew Tolmach. An external representation for the GHC core language. URL `citeseer.ist.psu.edu/tolmach01external.html`.

Mark Tullsen. First class patterns. In E. Pontelli and V. Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, January 2000.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium*

on Principles of Programming Languages, pages 307–312. Association for Computing Machinery, 1987. URL `citeseer.ist.psu.edu/wadler86views.html`.

Philip Wadler. The expression problem. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`, 1998.

Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press. URL `citeseer.ist.psu.edu/wallace99haskell.html`.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003. URL `http://doi.acm.org/10.1145/640128.604150`.

A. Full Versions of `dataToExpQ` and `dataToPatQ`

```
dataToQa mkCon mkLit appCon antiQ t =
  case antiQ t of
    Nothing →
        case constrRep constr of
            AlgConstr _ →
                appCon con conArgs
            IntConstr n →
                mkLit $ TH.integerL n
            FloatConstr n →
                mkLit $ TH.rationalL (toRational n)
            StringConstr (c : _) →
                mkLit $ TH.charL c
      where
        constr :: Constr
        constr = toConstr t
        constrName :: Constr → String
        constrName k =
          case showConstr k of
                "(:)" → ":"
                name → name
        con      = mkCon (TH.mkName (constrName constr))
        conArgs = gmapQ (dataToQa mkCon mkLit
                                  appCon antiQ)
                        t
    Just y → y
dataToExpQ :: Data a
              ⇒ (∀a.Data a ⇒ a → Maybe (TH.Q TH.Exp))
              → a
              → TH.Q TH.Exp
dataToExpQ = dataToQa TH.conE TH.litE (foldl TH.appE)
dataToPatQ :: Data a
              ⇒ (∀a.Data a ⇒ a → Maybe (TH.Q TH.Pat))
              → a
              → TH.Q TH.Pat
dataToPatQ = dataToQa id TH.litP TH.conP
```