Direct Reflection for Free!

JOOMY KORKUT, Princeton University, USA

1 PROBLEM AND MOTIVATION

Haskell is considered to be one of the best in class for language implementations [9]. It has been the metalanguage of choice for production-ready languages such as Elm, PureScript and Idris, proof of concept implementations such as Pugs (of Perl 6), and many toy languages. However, adding a metaprogramming system, even for toy languages, is a cumbersome task that makes maintenance costly. Both implementing a metaprogramming feature in the first place and keeping it updated to work with any change to the language or the abstract syntax tree (AST) is costly, since that feature would depend on the shape of the entire AST — namely quasiquotation [8] is such a feature. For example, the Idris compiler [4] suffers from this problem: Its implementation of quotation and unquotation is 1200 lines of Haskell, accompanied by 2500 lines of Idris library code to make that work, and most of this is boilerplate code.

In order to solve this problem we look for ways to augment existing language implementations with metaprogramming facilities automatically. If you have evaluation, you should be able to evaluate quasiquoted terms for free. If you have type-checking, you should be able to type-check quasiquoted terms for free. If you have a parser, you should have parser reflection for free.

2 BACKGROUND AND RELATED WORK

For almost two decades, Haskell has been equipped with the Scrap Your Boilerplate [11, 12] style generic programming, which lets users traverse abstract data types with less boilerplate code. In modern Haskell, this style is embodied by the `Data` and `Typeable` type classes, which can be derived automatically. These type classes allow representation of types as run-time values, and inspection of constructors and constructor fields. We leverage this mechanism to achieve our goal.

There is little work on adding metaprogramming facilities automatically to an existing language. Berger et al. [3] describe a calculus that models both compile-time and run-time homogeneous generative metaprogramming, and give a recipe for adding metaprogramming to any language. Their recipe involves creating a new syntactic form for AST terms, while we will take a different approach in this work. We encode ASTs of a language in the same language instead. Furthermore, not all metaprogramming is generative, hence we find a method that enables metaprogramming methods that involve breaking down existing data type and function definitions.

Encoding ASTs in the same language is a well known concept. Especially research in self-evaluation has generated ways to represent many languages in themselves. The Mogensen encoding [15] is an encoding of untyped λ-calculus in itself; it combines the Scott encoding [1] with higher-order abstract syntax [16]. A paper by Stump [19] explains the design decisions behind a self-representing and self-evaluating language that is more expressive about variable names. On the implementation side of untyped languages, Lisp's quotation mechanism and meta-circular evaluator [14] have inspired further research on metaprogramming in general.

Research on self-evaluation is not limited to untyped languages, there is a long line of work by Brown and Palsberg [5–7] that defines encodings for languages like System U and System F_ω. However, encoding a language in itself does not need to self-evaluate. Template Haskell [17] is an example of a compile-time metaprogramming system, where Haskell is represented in itself. The pattern we describe in this work can be used with any of these encodings.

Author's address: Joomy Korkut, Princeton University, Princeton, New Jersey, USA, joomy@cs.princeton.edu
ACM member number: 3665406. Entry for ICFP'19 SRC, Graduate Category. Advised by Andrew W. Appel.

3 APPROACH AND UNIQUENESS

In this work, we will present a pattern that takes advantage of generic programming when we implement toy programming languages in Haskell and decide to add metaprogramming features. We will show that this pattern can be used to convert any Haskell type (with an instance of the type classes above), back and forth with our all-time favorite toy language: λ-calculus. We *reify* Haskell values into their Scott encodings [1] in the λ-calculus AST, and *reflect* them back to Haskell values. This conversion can then be used to add metaprogramming features to the languages we implement in Haskell, via *direct reflection*, a technique that makes the language implementation in the metalanguage a part of the object language's semantics [2].

Our goal is to capture the representation of object language syntax trees within the same object language. Initially, it helps to generalize this into the conversion of any metalanguage value into its encoding in the object language, whose AST is represented by the data type `Exp`. We define a type class that encapsulates this conversion in both directions:

```
class Bridge a where
  reify :: a -> Exp
  reflect :: Exp -> Maybe a
```

Once a `Bridge` instance is defined for a type `a`, we will have a way to `reify` it into a representation in `Exp`. However, if we only have an `Exp`, we can only recover a value of the type `a` if that `Exp` is a valid representation of some value.

For example, if our language contains a primitive type like strings, we can define an instance to declare how they should be converted back and forth:

```
instance Bridge String where
  reify s = StrLit s
  reflect (StrLit s) = Just s
  reflect _ = Nothing
```

The `reflect` function above states that if an expression is not a string literal, represented above with the constructor `StrLit`, it is not possible to recover a Haskell string from it.

3.1 λ-calculus and Scott encoding

For a Haskell value `C v_1` $\cdots$ `v_n` of type `T`, where `C` is the ith constructor out of m constructors of `T`, and `C` has arity n, the Scott encoding (denoted by $\lceil\ \rceil$) of this value will be

$$\lceil \texttt{C}\ v\_1 \cdots v\_n \rceil = \lambda c_1 \cdots c_m.\ (c_i\ \lceil v\_1 \rceil \cdots \lceil v\_n \rceil)$$

For a Haskell data type **data** `Color` = `Red` | `Green`, Scott encodings of the constructors will be

$$\lceil \texttt{Red} \rceil = \lambda c_1\ c_2.\ c_1 \qquad \lceil \texttt{Green} \rceil = \lambda c_1\ c_2.\ c_2$$

If we decide to add a new constructor `Blue` to the `Color` data type, we must update each of the Scott encodings above accordingly, so that we have:

$$\lceil \texttt{Red} \rceil = \lambda c_1\ c_2\ c_3.\ c_1 \quad \lceil \texttt{Green} \rceil = \lambda c_1\ c_2\ c_3.\ c_2 \quad \lceil \texttt{Blue} \rceil = \lambda c_1\ c_2\ c_3.\ c_3$$

When we implement λ-calculus in Haskell, we start by defining a data type for our AST, using Haskell strings for names:

```
data Exp = Var String | App Exp Exp | Abs String Exp
```

For metaprogramming, we need a representation of λ-calculus terms within λ-calculus, and a `Bridge` instance for `Exp` would achieve exactly that; it would give us an easy way to convert between the signified (ones we want to reify) and signifier terms (ones that are the result of reifying).

However, as we develop the language, we often need to add new constructors to the AST. If we define a `Bridge` instance now, and add more constructors to `Exp`, then the previous `Bridge` instance becomes obsolete. Suppose we want to add string literal, quasiquote and antiquote expressions:

```
data Exp = Var String | App Exp Exp | Abs String Exp
         | StrLit String | Quasiquote Exp | Antiquote Exp
```

How do we make sure that the `Bridge` instance does not become obsolete? The answer is to avoid defining a special `Bridge` instance for the `Exp` type. Ideally, we would like to have one *for free*, based on a different type class. This is where generic programming comes in. Using the `Data` and `Typeable` type classes, we define a `Data a => Bridge a` instance. Once defined, implementation of certain metaprogramming features via direct reflection becomes very easy. Now implementing quasiquotation is just a matter of adding two lines to the evaluation function:

```
eval (Quasiquote e) = reify e
eval (Antiquote e) = let Just e' = reflect (eval e) in e'
```

3.2 Typed λ-calculus and the sum-of-products encoding

The same pattern of defining a `Data a => Bridge a` instance with respect to an encoding of choice can be applied to typed languages as well. For the simplest example of this, we can look at simply typed λ-calculus with binary sums and products, unit and void types, and isorecursive (μ) types. The Haskell implementation for this language would need to have two main data types for its AST: `Ty` for types and `Exp` for terms.

```
data Ty = TyString | TyInt | Arr Ty Ty | TyUnit | TyVoid
        | Pair Ty Ty | Sum Ty Ty | Mu String Ty | TyVar String
```

In `Exp`, we introduce `Inl` and `Inr` constructors for the sum type, and `MkPair` for the product type. Using these types and constructors, we can encode Haskell data types in the sum-of-products style common in generic programming [13]. For example, the `Color` type from before would be encoded in this language as `Sum TyUnit (Sum TyUnit TyUnit)`, which is exactly 1 + 1 + 1, in a more common notation in type theory. A value of the type `Color` such as `Red` would be `Inl MkUnit`.

A recursive type such as `List Int` would be encoded as `Mu "x" (Sum TyUnit (Pair TyInt (TyVar "x")))`, which corresponds to $\mu x.1 + (\text{Int} \times x)$ in the common theoretical notation.

To implement these encodings for a given Haskell type, we have to augment the `Bridge` type class with a definition, `ty : Ty`. For this new `Bridge` type class, only the `Data a => Bridge a` instance is significantly different compared to the one for the untyped λ-calculus[1]. Once that instance is written, the implementation of quasiquotation is still only the same two lines.

4 RESULTS AND CONTRIBUTIONS

The design pattern described in this work allows automatic derivation of metaprogramming features from your language implementation. Our pattern works for both compile-time and run-time metaprogramming features. The features above are generative, but they also can be intensional features such as inspecting the context and taking apart function and data type definitions in and of the object language. This pattern can even be used to implement some form of computational reflection [10, 18], by reifying the context during run-time for a new special λ form. Metaprogramming implementations often require significant boilerplate code, and our work attempts to minimize that by using generic programming.

[1]The full code is in the repository: http://github.com/joom/direct-reflection-for-free

REFERENCES

[1] Martin Abadi, Luca Cardelli, and Gordon Plotkin. 1993. Types for the Scott numerals.

[2] Eli Barzilay. 2006. *Implementing reflection in Nuprl.* Ph.D. Dissertation. Cornell University.

[3] Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling Homogeneous Generative Meta-Programming. *ECOOP.*

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

[5] Matt Brown and Jens Palsberg. 2015. Self-Representation in Girard's System U. *POPL.*

[6] Matt Brown and Jens Palsberg. 2016. Breaking Through the Normalization Barrier: A Self-interpreter for F-omega. *POPL.*

[7] Matt Brown and Jens Palsberg. 2017. Typed Self-evaluation via Intensional Type Functions. *POPL.*

[8] David Raymond Christiansen. 2014. Type-directed elaboration of quasiquotations: a high-level syntax for low-level reflection. *IFL.*

[9] Gabriel Gonzalez. [n. d.]. State of the Haskell ecosystem. https://github.com/Gabriel439/post-rfc/blob/master/sotu.md. ([n. d.]). Accessed: 2019-01-30.

[10] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. 2008. Reflection for the Masses. In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Springer-Verlag, Berlin, Heidelberg, 87–122.

[11] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate - a practical design pattern for generic programming. *TLDI.*

[12] Ralf Lämmel and Simon Peyton Jones. 2005. Scrap your boilerplate with class: extensible generic functions. *ICFP.*

[13] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. *ACM Sigplan Notices* 45, 11 (2010), 37–48.

[14] John McCarthy and Michael I. Levin. 1965. *LISP 1.5 programmer's manual.* MIT press.

[15] Torben Æ. Mogensen. 1992. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming* 2 (1992), 345–364.

[16] Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. *PLDI.*

[17] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. *Haskell Workshop.*

[18] Brian Cantwell Smith. 1984. Reflection and semantics in LISP. *POPL.*

[19] Aaron Stump. 2009. Directly reflective meta-programming. *Higher-Order and Symbolic Computation* 22, 2 (2009), 115–144.