Direct Reflection for Free!

Joomy Korkut
Princeton University
joomy@cs.princeton.edu

Abstract

Haskell is considered to be one of the best in class when it comes to compiler development [4]. 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. Once any metaprogramming feature is implemented, every change to the abstract syntax tree (AST) may require that feature to be updated, since it would depend on the shape of the entire AST — namely quasiquotation [3].

For almost two decades, Haskell has been equipped with the Scrap Your Boilerplate [5] style of 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.

In this talk, I will present a trick that takes advantage of this method when we implement toy programming languages in Haskell and decide to add metaprogramming features. I will show that this trick 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 *reflect* Haskell values into their Scott encodings [1] in the λ-calculus AST, and *reify* them back to Haskell values. This conversion can then be used to add metaprogramming features via *direct reflection* [2], to the languages we implement in Haskell.

Abstraction

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
  reflect :: a -> Exp
  reify :: Exp -> Maybe a
```

Once a `Bridge` instance is defined for a type `a`, we will have a way to `reflect` 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 the 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
  reflect s = StrLit s
  reify (StrLit s) = Just s
  reify _ = Nothing
```

The `reify` 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.

Application: λ-calculus and Scott encoding

For a Haskell value `C v_1 ⋯ v_n` of type `T`, where `C` is the ith constructor out of m constructors of `T`, and `C` has arity n, 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 reflect) and signifier terms (ones that are the result of reflecting).

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 | Quasiq Exp | Antiq 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 (Quasiq e) = reflect e
eval (Antiq e) = let Just e' = reify (eval e) in e'
```

In my talk, I will describe this trick and show its applications in typed languages such as simply-typed λ-calculus with μ-types and the calculus of constructions, to implement more complex metaprogramming features.

[1] Martin Abadi, Luca Cardelli, and Gordon Plotkin. 1993. Types for the Scott numerals. (1993).
[2] Eli Barzilay. 2006. *Implementing reflection in Nuprl.* Ph.D. Dissertation. Cornell University.
[3] David Raymond Christiansen. 2014. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *IFL*.
[4] 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.
[5] Ralf Lämmel and Simon Peyton-Jones. 2003. Scrap your boilerplate - a practical design pattern for generic programming. *TLDI*.