Direct Reflection for Free!

Joomy Korkut
Princeton University
Advised by Andrew W. Appel

If you have evaluation in your language...

you should be able to evaluate quasiquoted terms for free!

If you have type-checking in your language...

you should be able to type-check quasiquoted terms for free!

If you have a parser for your language...

you should have parser reflection for free!

Problem and Motivation

Haskell is considered to be one of the best in class for language implementations. It has been the metalanguage of choice for languages such as Elm, PureScript and Idris, 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** is such a feature.

For example, the Idris compiler 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 present a design pattern to augment existing language implementations with metaprogramming facilities automatically, using generic programming in the host language!

Here's the recipe!

1) Define an AST data type `Exp` for your language.

2) Define a type class to express conversion between a given Haskell type and expressions of the object language.

```
class Bridge a where
  reify :: a → Exp
  reflect :: Exp → Maybe a
  ty :: Ty
```

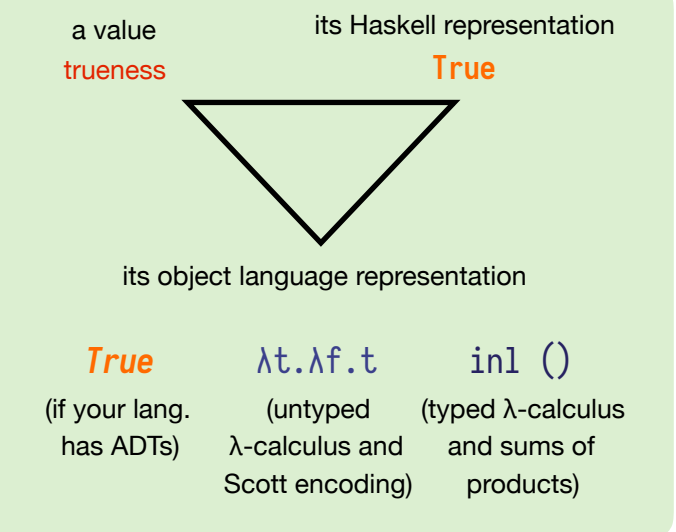
(where the last one occurs only for a typed language)

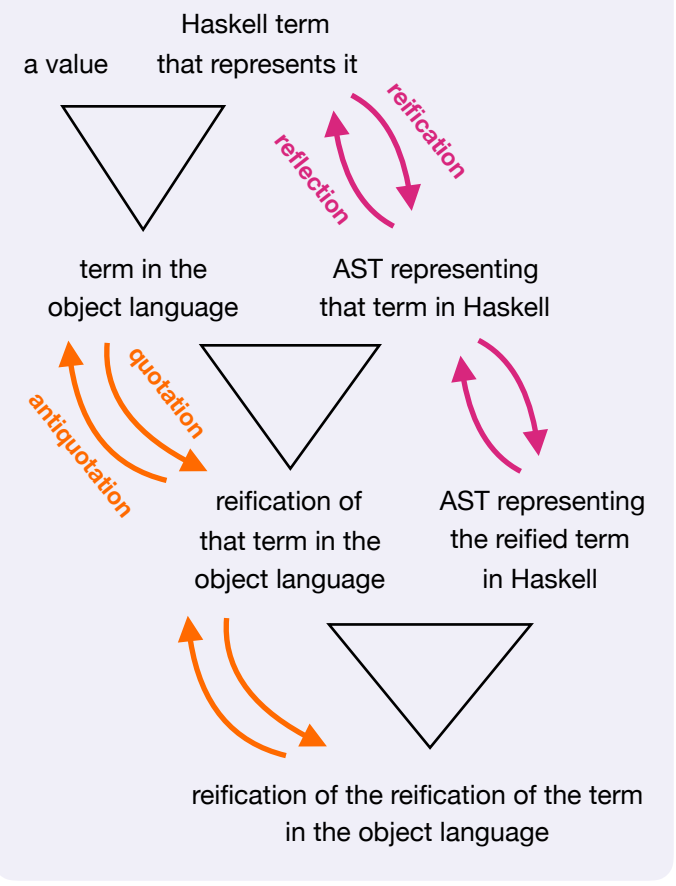
3) Pick a **self-representation** in order to capture the representation of object language syntax trees within the same object language.

- Scott encoding for untyped λ-calculus
- Sums of products for typed λ-calculus with sums, products and μ-types

4) Define a `Data a ⇒ Bridge a` instance based on that representation, to take advantage of Haskell's generic programming abilities.

5) Convert any Haskell term of a type with a `Data` instance to an object language term, and back. `Exp` itself is one of those types!





Trying the recipe on the **untyped** λ-calculus

We pick the Scott encoding as the way to represent Haskell terms in λ-calculus.

Here's how Scott encoding works in a nutshell:

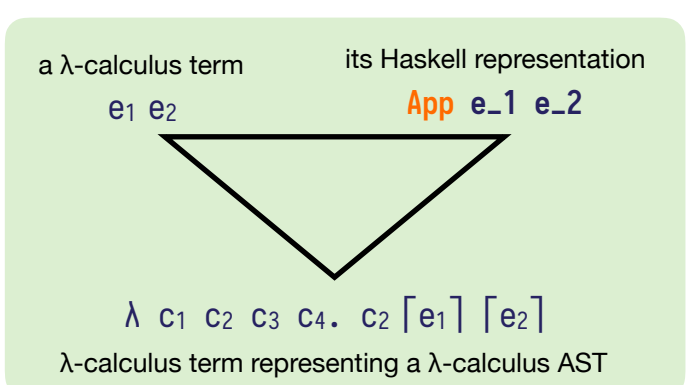
$$\lceil \texttt{Ctor}\ e_1\ \ldots\ e_n \rceil_{\text{Scott}} = \lambda\ c_1\ c_2\ \ldots\ c_m.\ c_i\ \lceil e_1 \rceil_{\text{Scott}}\ \ldots\ \lceil e_n \rceil_{\text{Scott}}$$

The idea is to add a λ binding for each constructor in the type, and constructing the body as an application of the picked constructor to the other Scott encoded subterms.

If we can encode any Haskell term in λ-calculus, we can also encode the `Exp` data type in λ-calculus, which allows us talk about object language ASTs in the object language!

```
data Exp = Var String | App Exp Exp
         | Abs String Exp | StrLit String
  deriving (Show, Eq, Data, Typeable)
```

We can write a `Data a ⇒ Bridge a` instance to do Scott encoding to a given Haskell value. `Data` empowers us to learn the necessary information!



Trying the recipe on the **typed** λ-calculus

For the simply typed λ-calculus extended with sums, products and μ-types, we pick sums of products as the way to represent Haskell terms in the object language.

Here are some examples of how this encoding would work:

$$\lceil \texttt{Bool} \rceil_{\text{SoP}} = 1 + 1$$

$$\lceil \texttt{Nat} \rceil_{\text{SoP}} = \mu\ N.\ 1 + N$$

$$\lceil \texttt{List Nat} \rceil_{\text{SoP}} = \mu\ L.\ 1 + \lceil \texttt{Nat} \rceil_{\text{SoP}} \times L$$

The types on the right would be represented by this AST type for object language types:

```
data Ty = TyUnit | TyVoid | TyString
        | Arr Ty Ty | Pair Ty Ty | Sum Ty Ty
        | TyVar String | Mu String Exp
  deriving (Show, Eq, Data, Typeable)
```

The encoding of Haskell values would look like this:

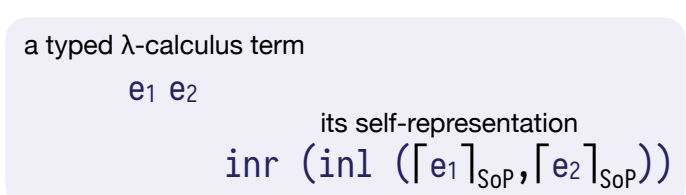
$$\lceil \texttt{True} \rceil_{\text{SoP}} = \texttt{inl()} \qquad \lceil \texttt{False} \rceil_{\text{SoP}} = \texttt{inr()}$$

$$\lceil \texttt{Z} \rceil_{\text{SoP}} = \texttt{inl()} \qquad \lceil \texttt{S Z} \rceil_{\text{SoP}} = \texttt{inr(inl())}$$

$$\lceil \texttt{(S Z) :: nil} \rceil_{\text{SoP}} = \texttt{inr}(\lceil \texttt{S Z} \rceil_{\text{SoP}}, \texttt{inl ()})$$

(assume equirecursive μ-types here for simple representation)

We can encode `Exp` terms in `Exp` as well!



Tying the knot

To complete the easy implementation of a metaprogramming feature, now all we have to do is to use `reify` and `reflect`, and apply whatever feature we want to implement in between.

Suppose we want to add **quasiquotation** and **antiquotation** to your language. We will start by extending our AST for these constructs:

```
data Exp = ...
         | Quasiquote Exp
         | Antiquote Exp
```

The code needed for evaluation of these constructs only takes a few lines, thanks to our design pattern!

```
eval' :: Map String Exp → Exp → Exp
...
eval' env (Quasiquote e) = reify e
eval' env (Antiquote e) =
   let Just x = reflect (eval e) in x
```

(we omit error handling here for brevity)

Now we can use our quasiquotation system and splice generated code in our programs.

```
λ> eval <$> parseExp "~( (λ x.x) `( () ) )"
Right MkUnit
```

splice · identity function · quote of unit

This feature can be used to implement parser reflection (similar to JavaScript's `eval`) or type-checker or elaborator reflection (similar to Idris)

Furthermore, implementing a kind of computational reflection by providing a new λ form with access to the context is also possible. Easy reuse of efficient host language code is another benefit of the pattern.