Direct Reflection for Free!

Joomy Korkut
Princeton University

advised by Andrew W. Appel

August 20th, 2019

ICFP 2019 Student Research Competition

Basic terminology

When we write an interpreter or a compiler, we are dealing with two languages:

- **Host language**: the language in which the interpreter/compiler is implemented.
- **Object language**: the input language of the generated interpreter/compiler.
- Examples:
  host: OCaml, object: Coq
  host: Haskell, object: Agda

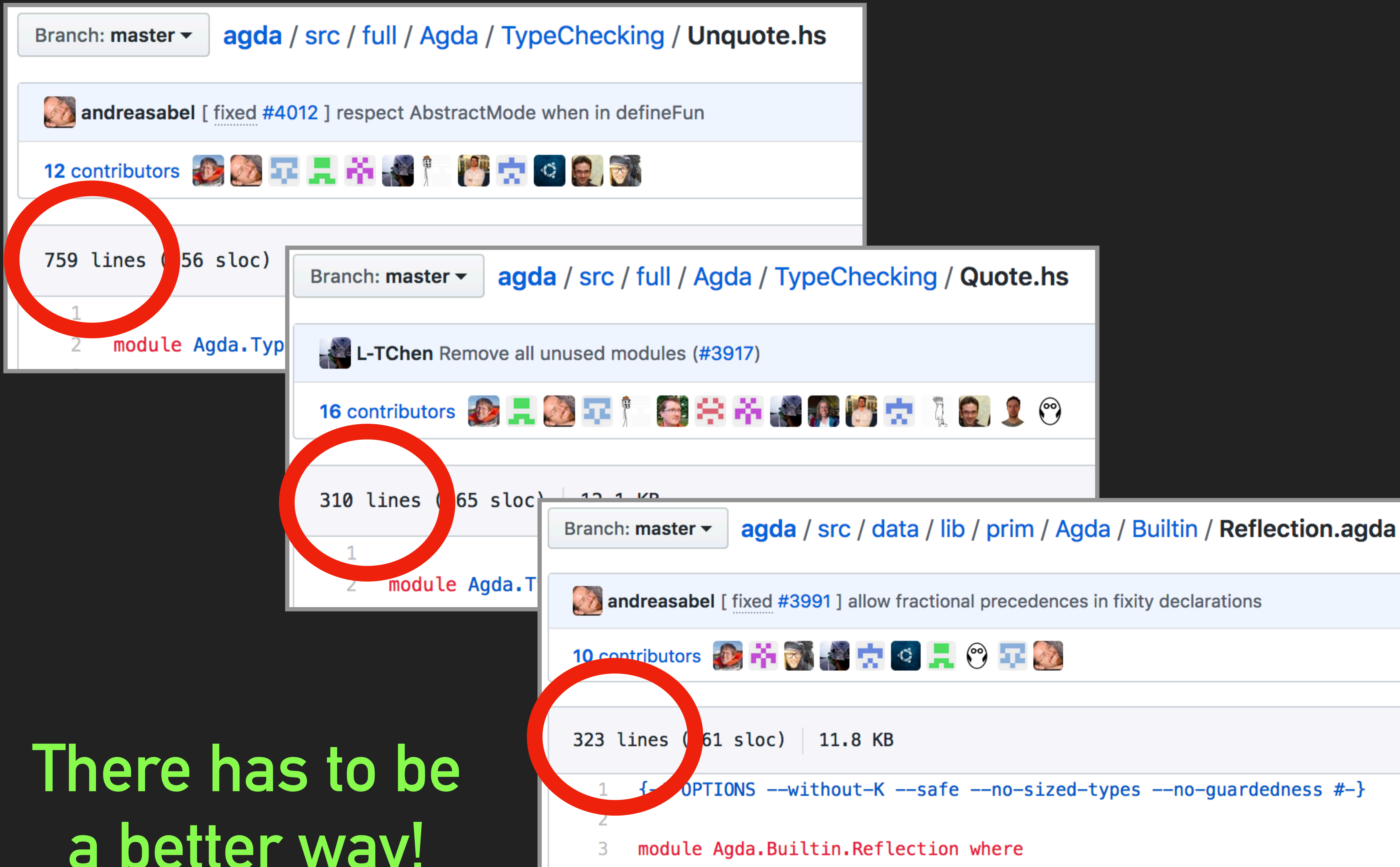
Basic terminology

- Metaprogramming is treating program fragments as data.
- We want to inspect these program fragments and generate new program fragments.
- We also want to run these program fragments as actual programs! (splice or unquote or antiquote)

Problem and Motivation

- Implementing metaprogramming systems, when writing a compiler/interpreter, is difficult.
- It's hard to maintain!
- Even for stable languages, these implementations are loooooooooooong.


Branch: master
Idris-dev / src / Idris / Reflection.hs
ahmadsalim Remove old eliminator generation and induction tactic (#4351)
12 contributors
1231 lines
Branch: master
Idris-dev / libs / prelude / Language / Reflection.idr
joom Add Bool, Maybe, Either implementations for Quotable (#4367)
7 contributors
1288 lines (1074 slo
Branch: master
Idris-dev / libs / prelude / Language / Reflection / Elab.idr
joom Add tryCatch as an Elab action in Language.Reflection.Elab
8 contributors
776 lines (648 sloc) 25.2 KB
1 ||| Primitives and tactics for elaborator reflection.
||| Elaborator reflection allows Idris code to control Idris's
4 ||| built-in elaborator, and re-use features like the unifier, the
5 ||| type checker, and the hole mechanism.
6 module Language.Reflection.Elab



There has to be a better way!

My solution

- Use the generic programming abilities of the host language, to derive a metaprogramming feature for the object language.
- This significantly shortens the code needed.
- It is automatically up to date with the AST.

In other words...

- If you have evaluation for your language, you should be able to evaluate quasiquoted terms for free!
- If you have type-checking for your language, you should be able to type-check quasiquoted terms for free!
- When you automate conversion between Haskell terms and object language terms, you can reuse your Haskell functions!

Here's the recipe!

me

1. Pick your object language. (What language do you want to implement?)

2. Define AST data types in Haskell for your object language. (`Exp`, `Ty`, `Pat`, whatever)

3. Pick a representation method.
   - Scott encoding for the untyped λ-calculus
   - Sums of products for the typed λ-calculus

4. Define a `Bridge` type class for your language.

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

Here's the recipe!

me

1. Pick your object language. (What language do you want to implement?)

2. Define AST data types in Haskell for your object language. (`Exp`, `Ty`, `Pat`, whatever)

3. Pick a representation method.
   Scott encoding for the untyped λ-calculus
   Sums of products for the typed λ-calculus

4. Define a `Bridge` type class for your language.

5. Define a `Data a ⇒ Bridge a` instance for the AST data type.

Here's the recipe!

1. Pick your object language. (What language do you want to implement?)

2. Define AST data types in Haskell for your object language. (`Exp`, `Ty`, `Pat`, whatever)

3. Pick a representation method.
   Scott encoding for the untyped λ-calculus
   Sums of products for the typed λ-calculus

4. Define a `Bridge` type class for your language.

5. Define a `Data a ⇒ Bridge a` instance for the AST data type.

6. Profit!

```
data Exp =
    Var String              x
  | App Exp Exp             e1 e2
  | Abs String Exp          λ x. e
  | StrLit String           "hello"
  | MkUnit                  ( )
 deriving (Show, Eq, Data, Typeable)
```

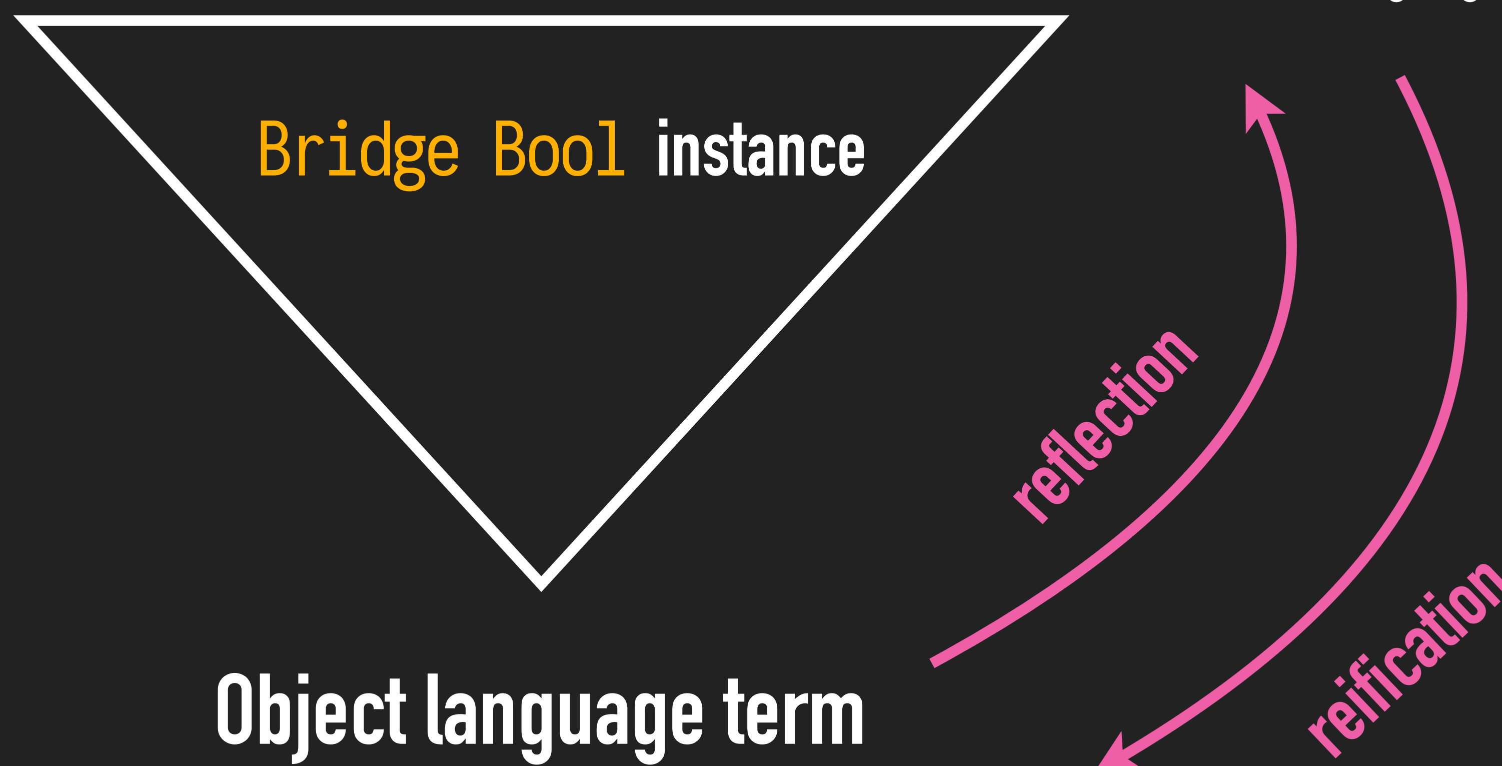
The Haskell terms triangle

A value

trueness

Haskell term
that represents it

`True`
(in meta language)



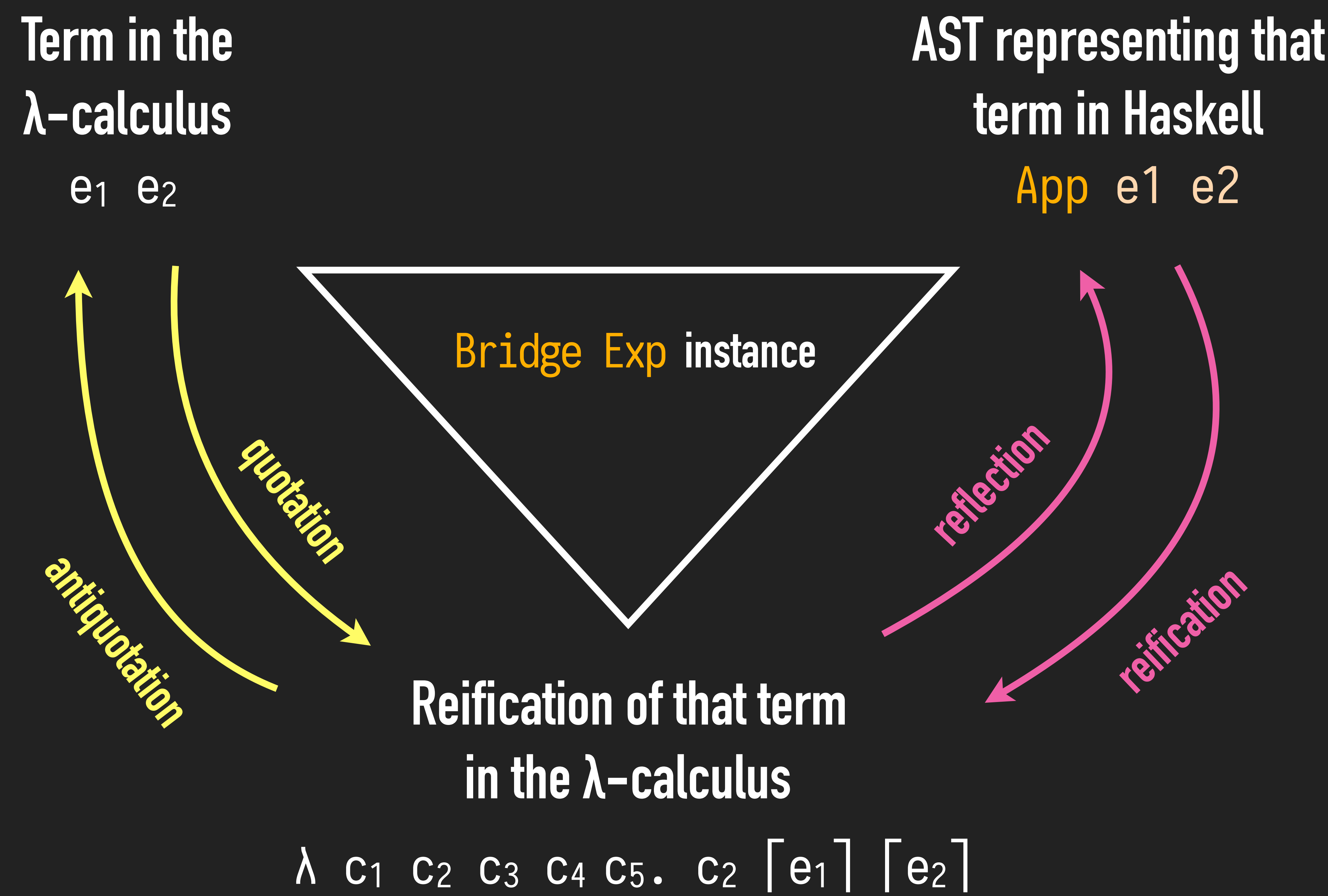
Object language term
that represents it

`True`
(if our object language has ADTs)

`λt.λf.t`
(if our object language is untyped λ-calculus)

`inl ()`
(if our object language is typed λ-calculus with sums and products)

The meta values triangle



```haskell
data Exp =
    Var String              x
  | App Exp Exp             e1 e2
  | Abs String Exp          λ x. e
  | StrLit String           "hello"
  | MkUnit                  ( )
  | Quasiquote Exp          `(e)
  | Antiquote Exp           ~(e)
  deriving (Show, Eq, Data, Typeable)
```

Tying the knot

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

(no error handling here)


Branch: master
Idris-dev / src / Idris / Reflection.hs
ahmadsalim Remove old eliminator generation and induction tactic (#4351)
12 contributors
1231 lines
Branch: master
Idris-dev / libs / prelude / Language / Reflection.idr
joom Add Bool, Maybe, Either implementations for Quotable (#4367)
7 contributors
1288 lines
Branch: master
Idris-dev / libs / prelude / Language / Reflection / Elab.idr
joom Add tryCatch as an Elab action in Language.Reflection.Elab
8 contributors
776 lines
25.2 KB
||| Primitives and tactics for elaborator reflection.
|||
||| Elaborator reflection allows Idris code to control Idris's
||| built-in elaborator, and re-use features like the unifier, the
||| type checker, and the hole mechanism.
module Language.Reflection.Elab

Tying the knot

in the Haskell REPL

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

identity function

quoting unit

splicing the function application

```
"`( () )"
```

concrete syntax of our object language

What else can we achieve using this pattern?

- Type checker / elaborator reflection: a way to expose the type-checker in the object language and make it available for the reflected terms, usable in metaprograms.

- Inspecting the context in runtime by reifying and reflecting the context, giving us a kind of computational reflection

- Reuse of efficient host language code by adding object language primitives

Extra slides

"In programming languages, there is a simple yet elegant strategy for implementing reflection: instead of making a system that describes itself, the system is made available to itself. We name this direct reflection, where the representation of language features via its semantics is actually part of the semantics itself."

Eli Barzilay, PhD dissertation, 2006

Generalizing Scott encoding

```
⌜Ctor e_1 ... e_n⌝
```

(in meta language)

=

```
λ c_1. λ c_2. ... λ c_m. c_i ⌜e_1⌝ ... ⌜e_n⌝
```

where `Ctor` is the ith constructor
out of m constructors

Key idea: if `Ctor` constructs a value of a type that has a `Data` instance, then we can get the Scott encoding automatically

Haskell's generic programming techniques

There are a few alternatives such as GHC.Generics, but I chose `Data` and `Typeable` for their expressive power.

```haskell
class Typeable a where
  typeOf :: a -> TypeRep
```

```haskell
class Typeable a => Data a where
  ...
  toConstr :: a -> Constr
  dataTypeOf :: a -> DataType
```

```haskell
gmapQ :: (forall d. Data d => d -> u) -> a -> [u]
```

(can collect arguments of a value)

```haskell
fromConstrM :: forall m a. (Monad m, Data a) => (forall d. Data d => m d) -> Constr -> m a
```

(monadic helper to construct new value from constructor)

Both `Data` and `Typeable` are automatically derivable! (for simple Haskell ADTs)

Implementation of Scott encoding from `Data`

```haskell
instance Data a => Bridge a where
  reify v
    | getTypeRep @a == getTypeRep @Int = reify @Int (unsafeCoerce v)
    | getTypeRep @a == getTypeRep @String = reify @String (unsafeCoerce v)
    | otherwise =
      lams args (apps (Var c : gmapQ reifyArg v))
   where
    (args, c) = constrToScott @a (toConstr v)
    reifyArg :: forall d. Data d => d -> Exp
    reifyArg x = reify @d x

  reflect e
    ...
```

(hack)

1. get all the constructors
2. pick which one you use
3. recurse on the arguments
4. construct the nested lambdas and applications

Implementation of Scott encoding from Data

```haskell
instance Data a ⇒ Bridge a where
  reify v
    ...

  reflect e
      | getTypeRep @a == getTypeRep @Int = unsafeCoerce (reflect @Int e)
      | getTypeRep @a == getTypeRep @String = unsafeCoerce <$> (reflect @String e)
      | otherwise =
     case collectAbs e of -- dissect the nested lambdas
       ([], _) → Nothing
       (args, body) →
        case spineView body of -- dissect the nested application
          (Var c, rest) → do
            ctors ← getConstrs @a
            ctor ← lookup c (zip args ctors)
            evalStateT (fromConstrM (reflectArg ctor)) rest
          _ → Nothing
    where
      reflectArg :: forall d. Data d ⇒ StateT [Exp] Maybe d
      reflectArg = do e ← gets head
                      modify tail
                      lift (reflect @d e)
```

(hack)

1. get the nested lambda bindings
2. get the head of the nested application
3. recurse on the arguments
4. construct the Haskell term

What we can do using this

- **Parser reflection**: a way to pass a string containing code in the object language, to the object language, and getting the reflected term.
- **Type checker / elaborator reflection**: a way to expose the type checker in the object language and make it available for the reflected terms, usable in metaprograms.
- **Reuse of efficient host language code**

Future work

- More experiments with typed object languages, especially dependent types
- Boehm-Berarducci encoding
- Object languages with algebraic data types
- Typed metaprogramming à la Typed Template Haskell or Idris
- Another metalanguage: Coq, JavaScript?

Related Work

- We did not have a convincing way to automatically add homogeneous generative metaprogramming to an existing language definition, until "Modelling Homogeneous Generative Meta-Programming" by Berger, Tratt and Urban (ECOOP'17)

  However, their one-size-fits-all method requires the addition of a new constructor to the AST to represent ASTs. And the addition of "tags" as well.

- We still do not have a convincing way to automatically add homogeneous generative metaprogramming to an existing language implementation.