Direct Reflection for Free!

Joomy Korkut
Princeton University
@cattheory

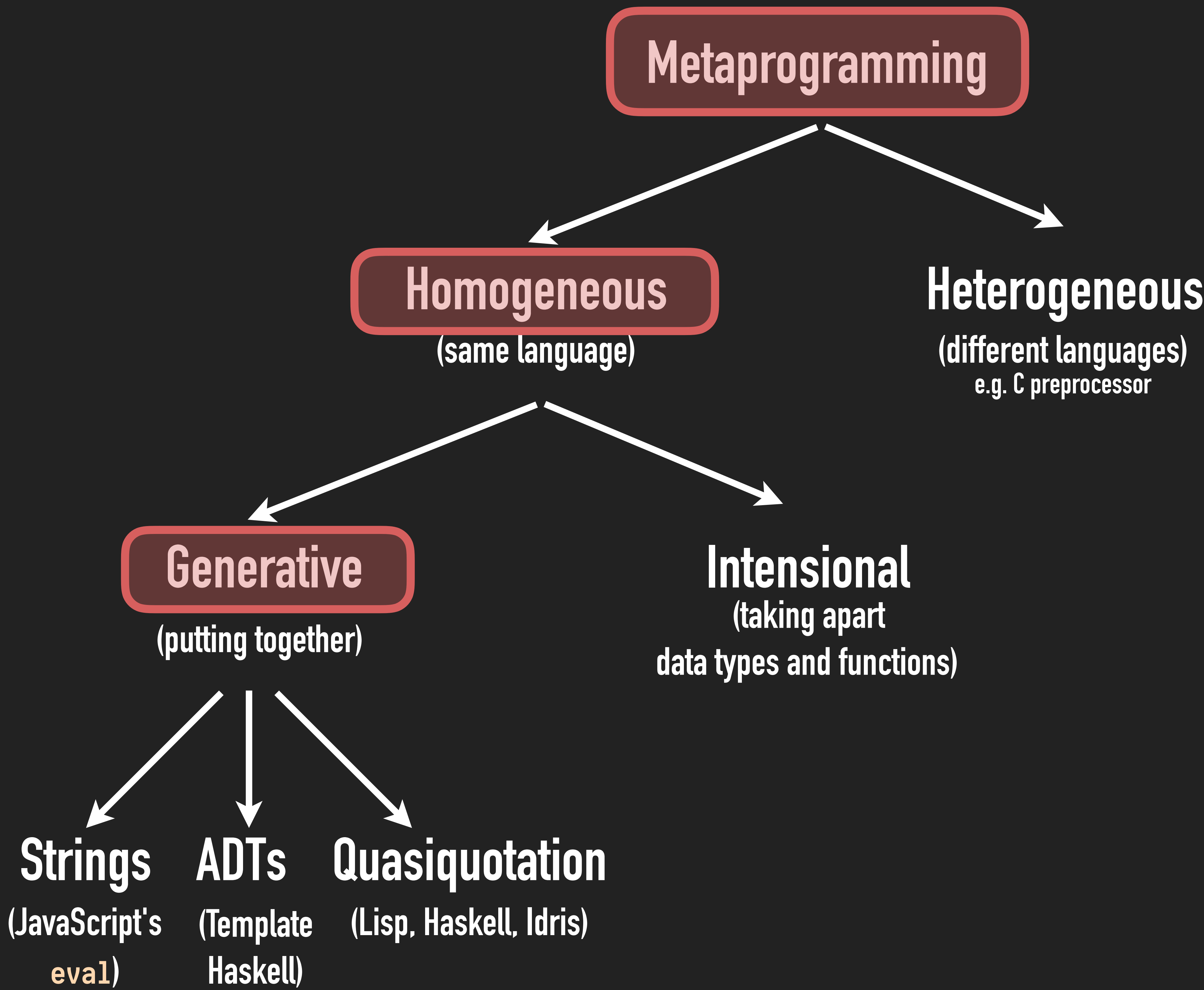
February 25th, 2019

NYPLSE '19

Basic terminology

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

- Metalanguage: the language in which the interpreter/compiler is implemented.
- Object language: the input language of the generated interpreter/compiler.



categorization from Martin Berger's 2016 slides

Problem statement

- Implementing metaprogramming systems, when writing a compiler/interpreter, is difficult. Especially with languages in development, any change in the language will require a lot of work to keep the metaprogramming parts up to date.

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

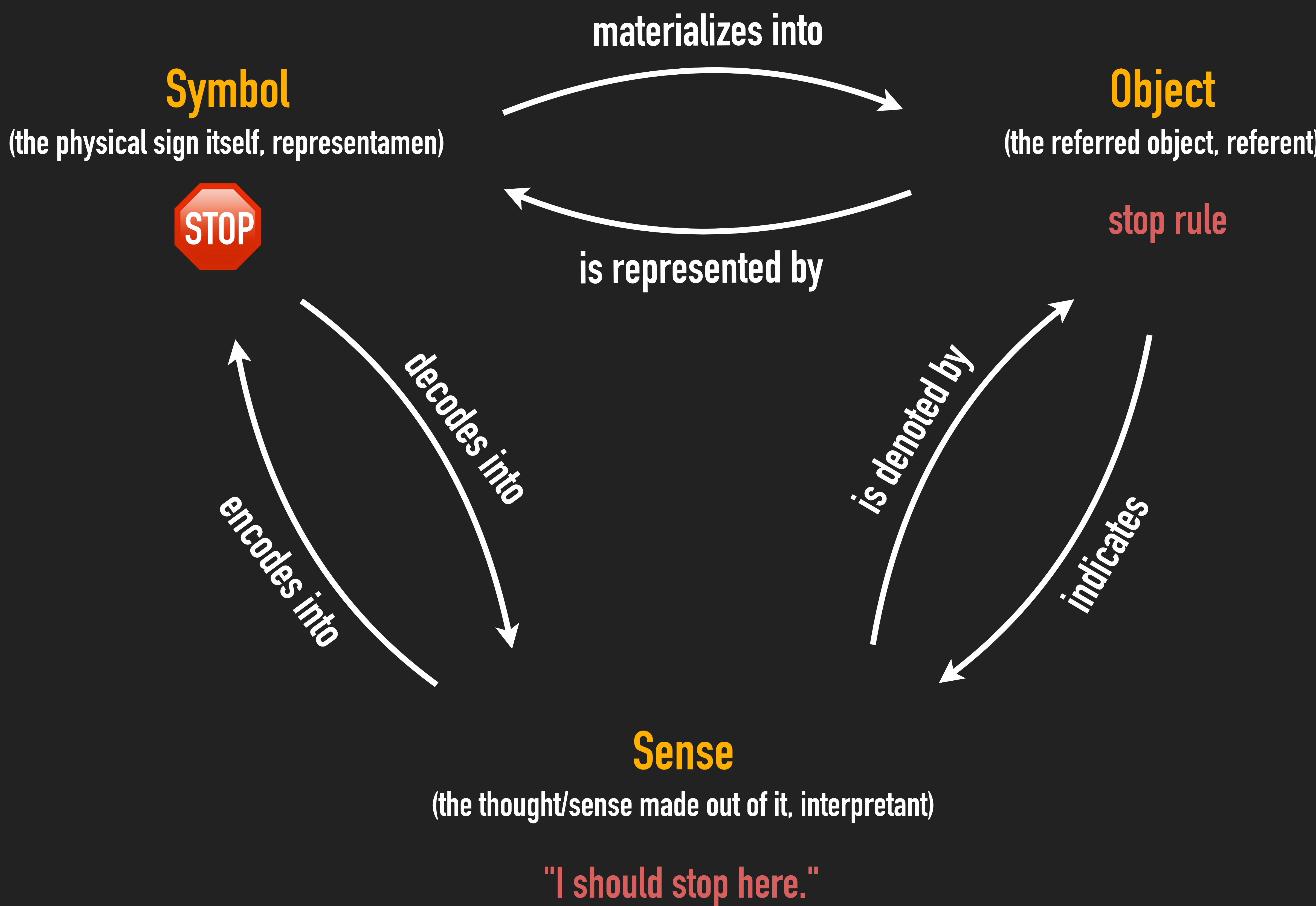
  However, their one-size-fits-all method requires the addition of a new constructor to the AST to represent ASTs. And the addition of "tags" as well.

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

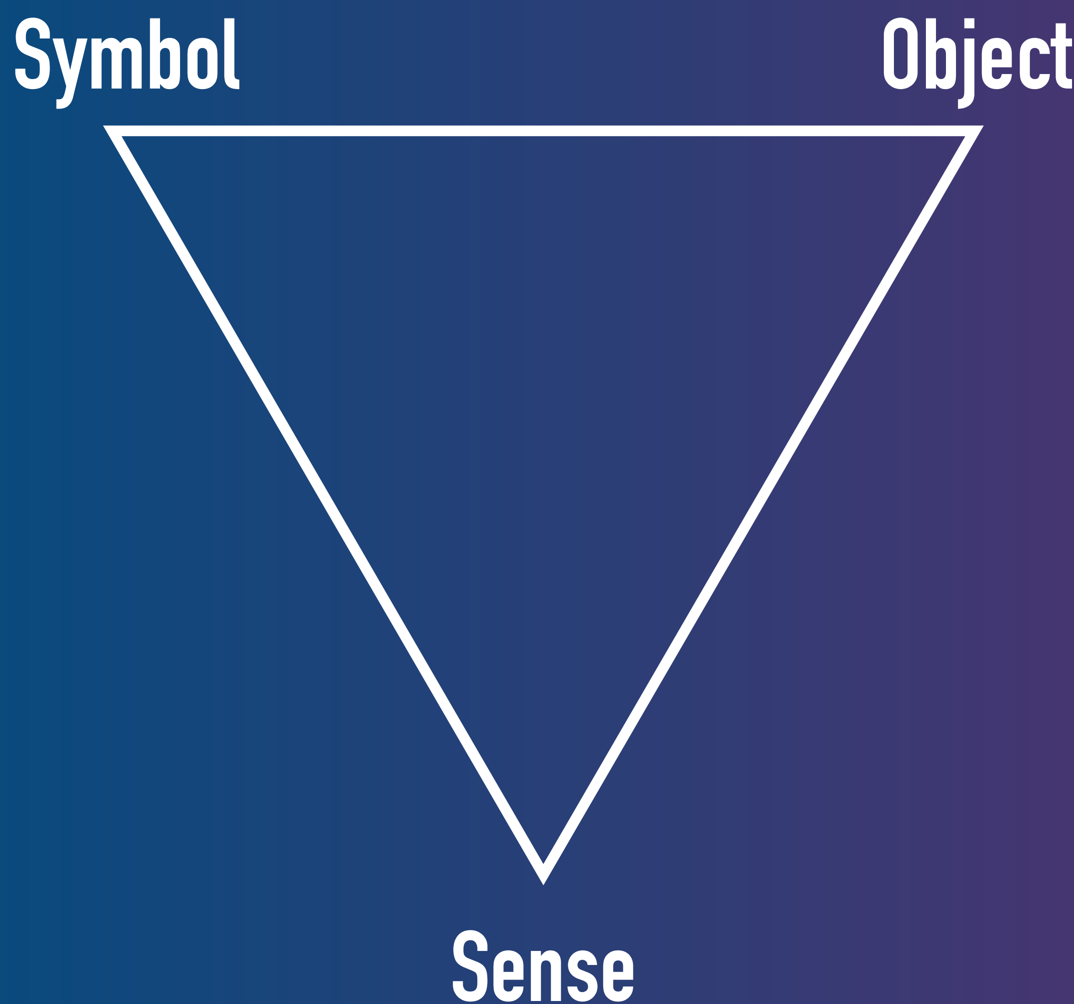
My solution

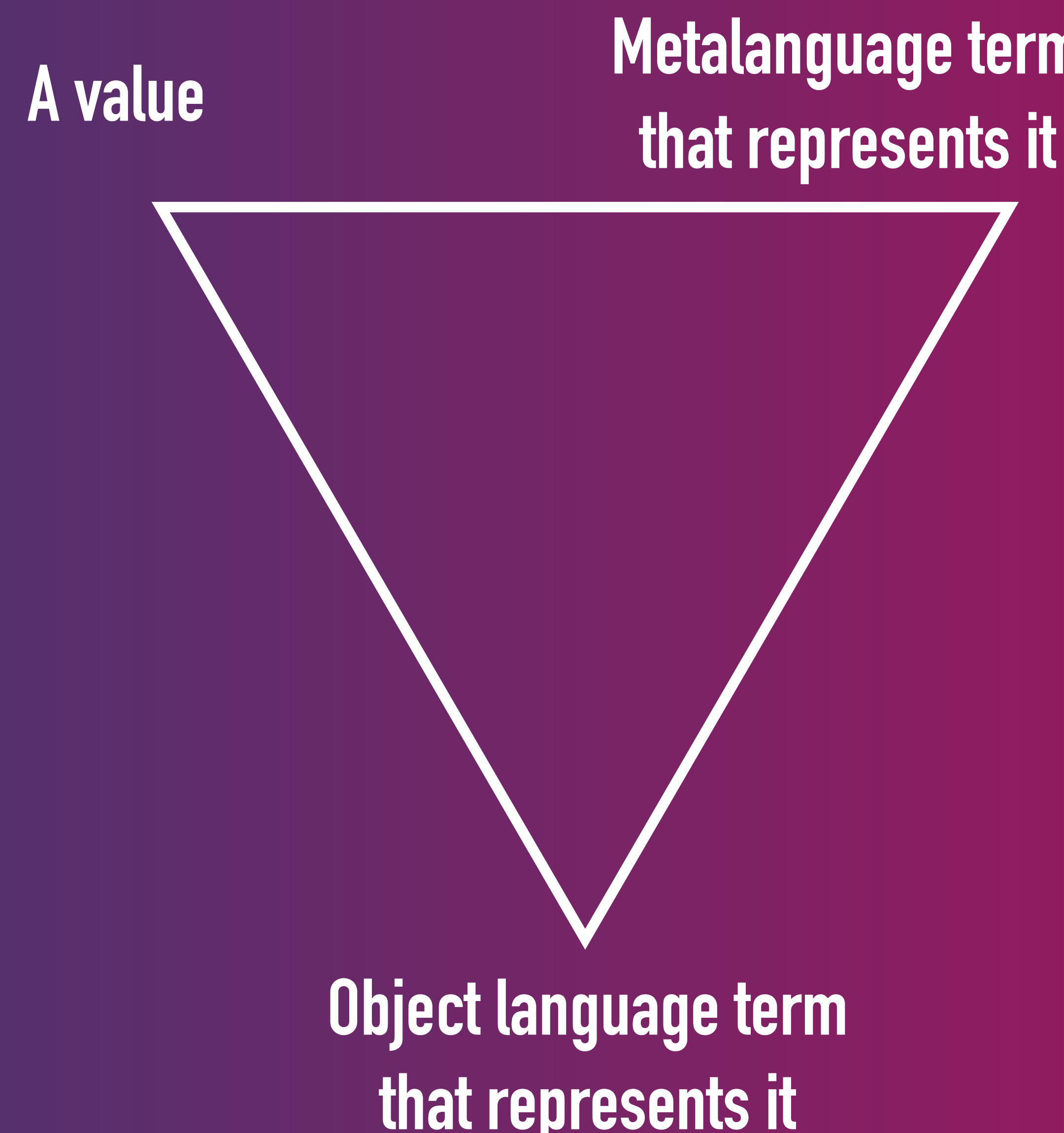
- To find an appropriate representation of ASTs of an object language inside that language. We can pick a different representation for each language.
- To use Haskell and take advantage of the generic programming techniques to automatically add metaprogramming to an existing language implementation.
- In other words, I want to use the intensional metaprogramming of the meta language to automatically create a generative metaprogramming system for the object language.

Peirce's triangle of signs



Peirce's triangle of signs, with a twist





(in a language implementation)

inspired from James Noble and Kumiko Tanaka-Ishii

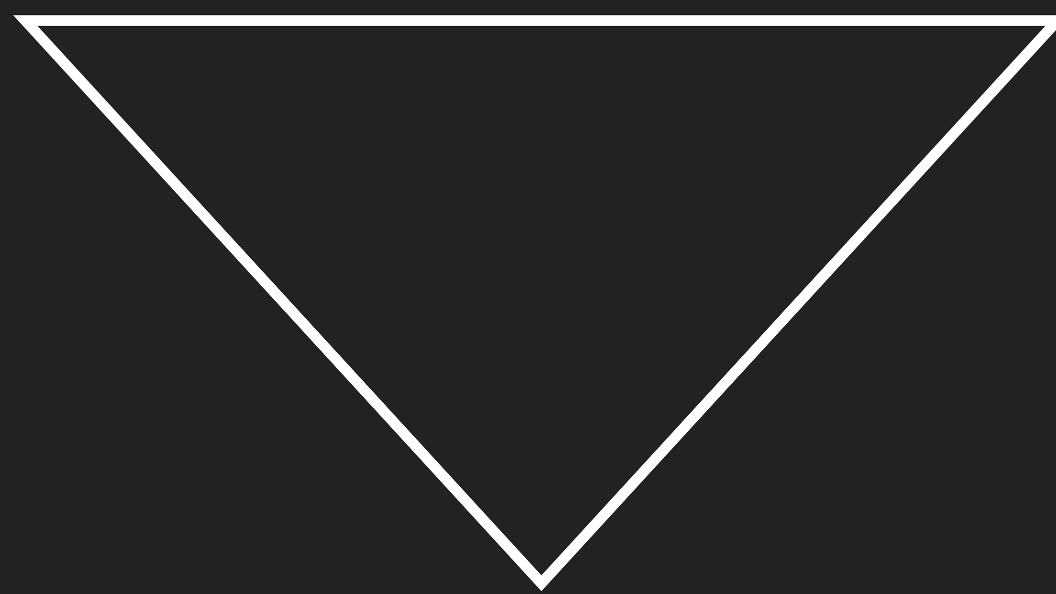
The language implementation triangle

A value

the mathematical value
red

**Meta language term
that represents it**

Red
(in meta language)



**Object language term
that represents it**

Red
(if our object language
has algebraic data types)

`λr.λg.λb.r`
(if our object language
is untyped λ-calculus)

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

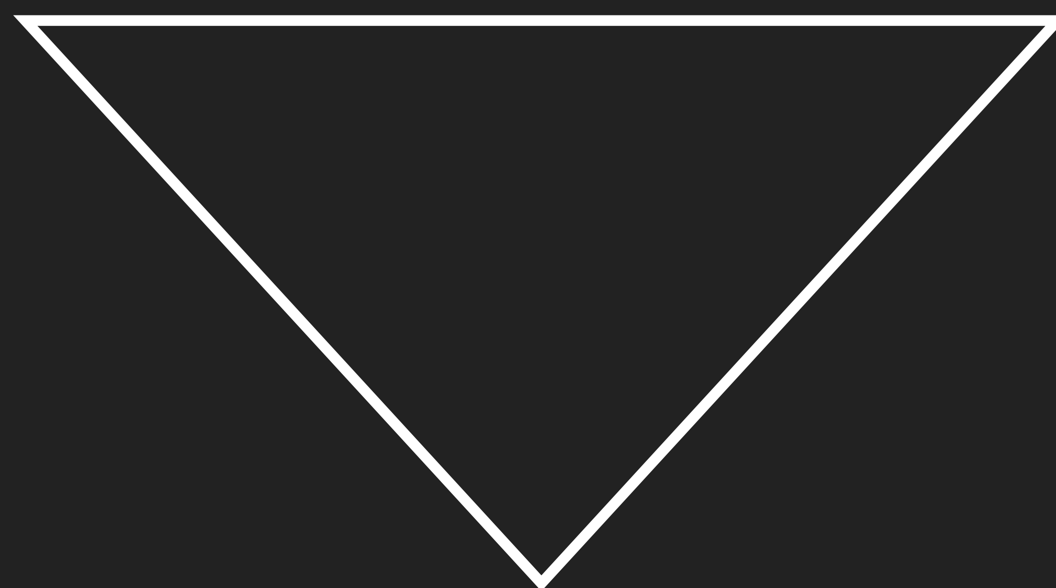
The language implementation triangle

A value

the string hello

Meta language term
that represents it

`"hello"`
(in meta language)

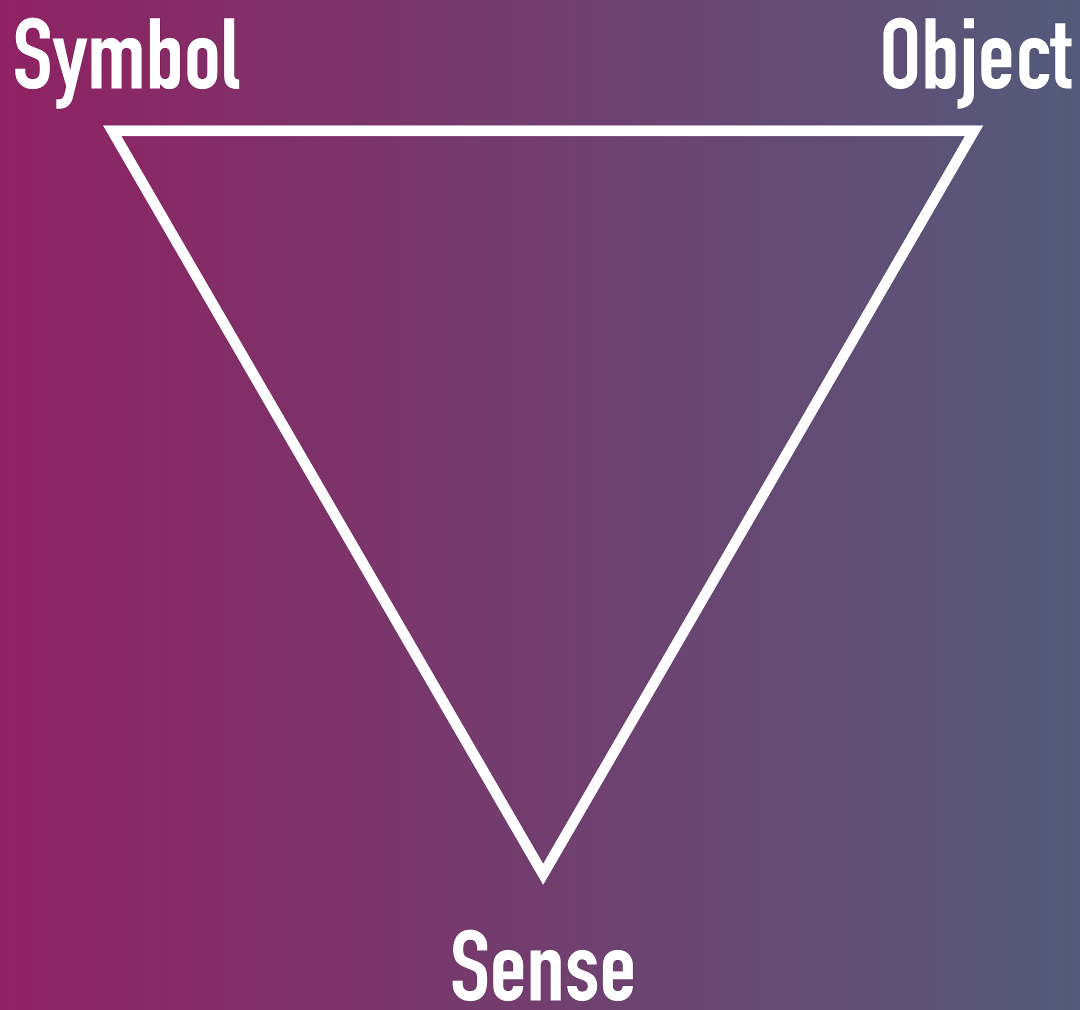


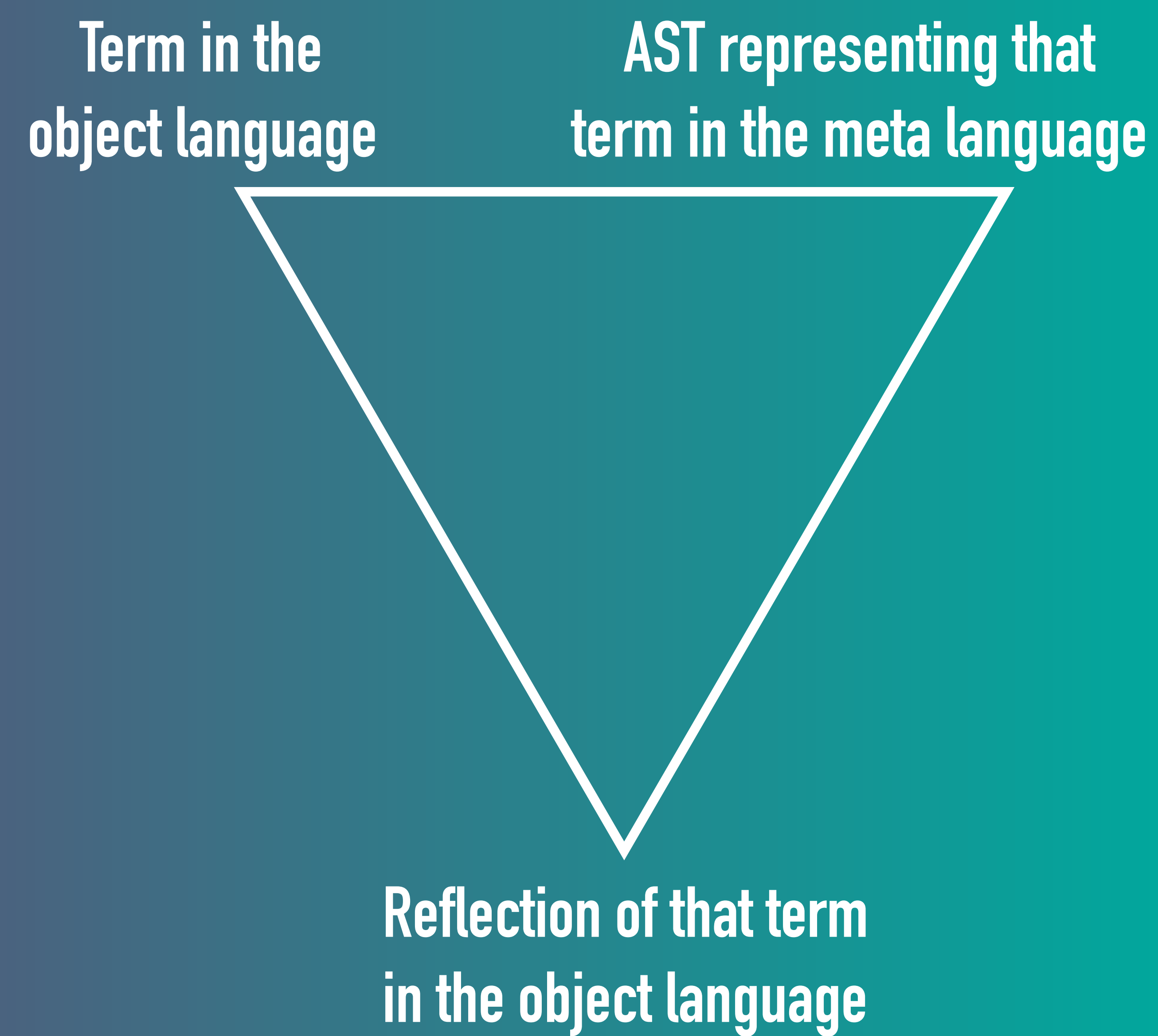
Object language term
that represents it

`"hello"`
(if our object language
has strings)

any other representation
our object language supports

Peirce's triangle of signs, with another twist





(in a language implementation)

The metaprogramming implementation triangle

Term in the object language

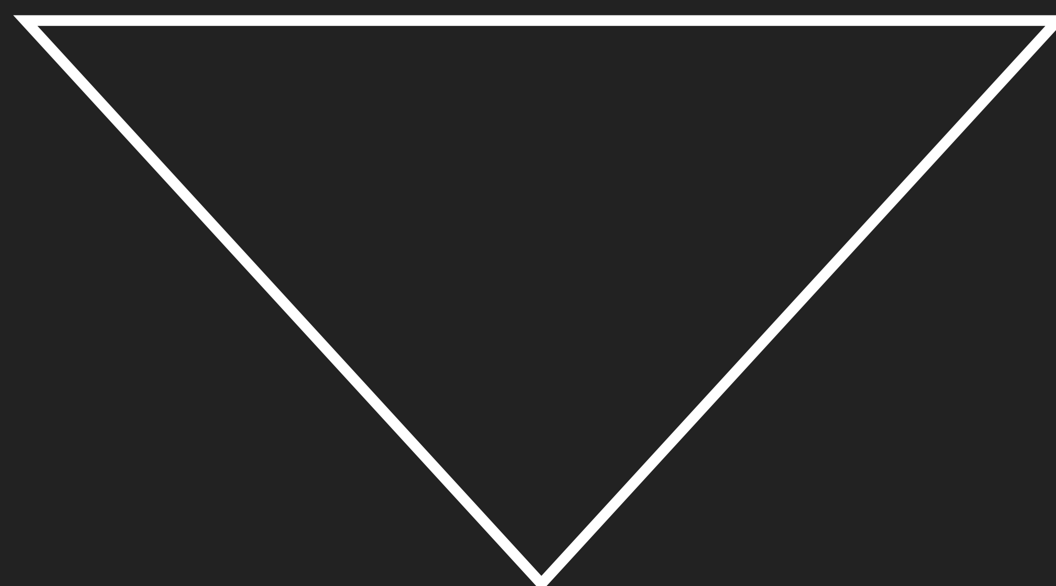
`"hello"`

(in object language)

AST representing that term in the meta language

`StrLit "hello"`

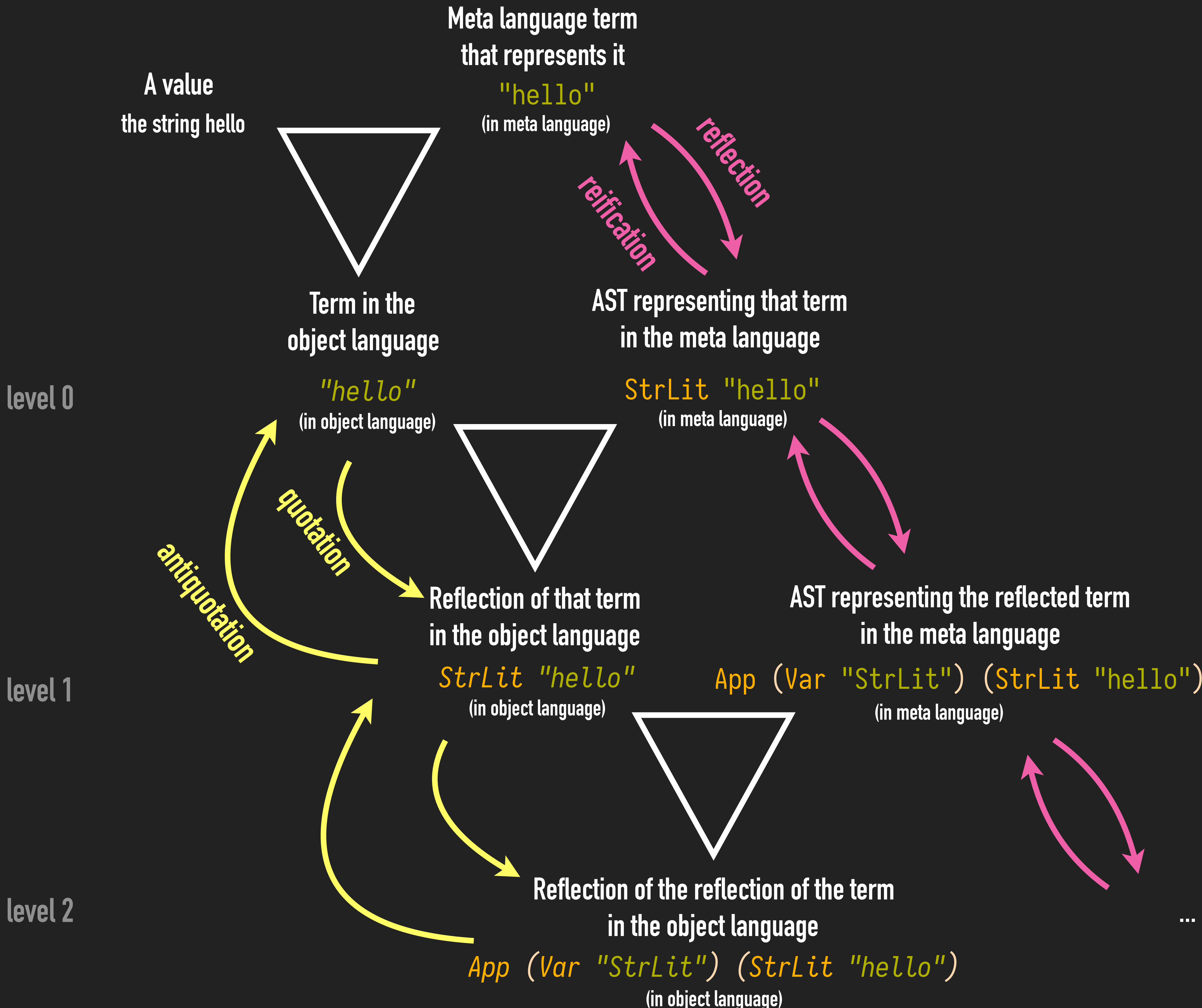
(in meta language)



Reflection of that term in the object language

`StrLit "hello"`

(in object language)


A value
the string hello
Meta language term
that represents it
"hello"
(in meta language)
reflection
reification
Term in the
object language
"hello"
(in object language)
AST representing that term
in the meta language
StrLit "hello"
(in meta language)
level 0
quotation
antiquotation
Reflection of that term
in the object language
StrLit "hello"
(in object language)
AST representing the reflected term
in the meta language
App (Var "StrLit") (StrLit "hello")
(in meta language)
level 1
level 2
Reflection of the reflection of the term
in the object language
App (Var "StrLit") (StrLit "hello")
(in object language)
...

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

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

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

instance Bridge Int where
  reflect n = IntLit n
  reify (IntLit n) = Just n
  reify _ = Nothing
```

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)

Cookbook 🧑‍🍳

1. Pick your object language.

2. Define an AST data type for your object language, in the metalanguage.

3. Pick your reflection representation.
   (There are many options!)

4. Define the `Data a => Bridge a` instance for the AST data type.

Let's try with the λ-calculus!

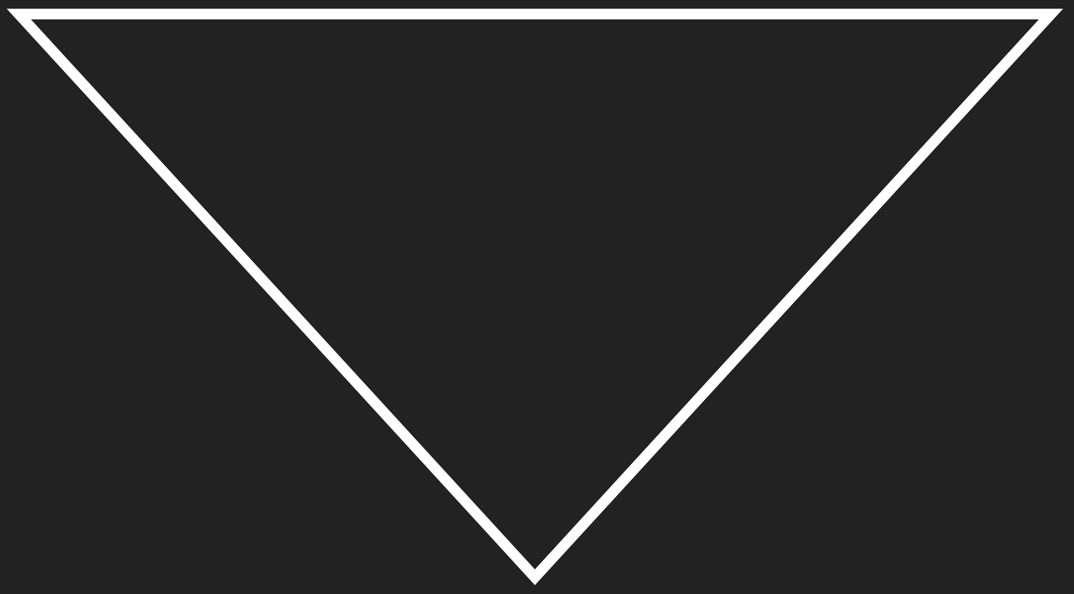
Scott encoding for untyped λ-calculus

A value

the natural number 0

Meta language term
that represents it

Z
(in meta language)



Object language term
that represents it

`λf.λx. x`

Scott encoding for untyped λ-calculus

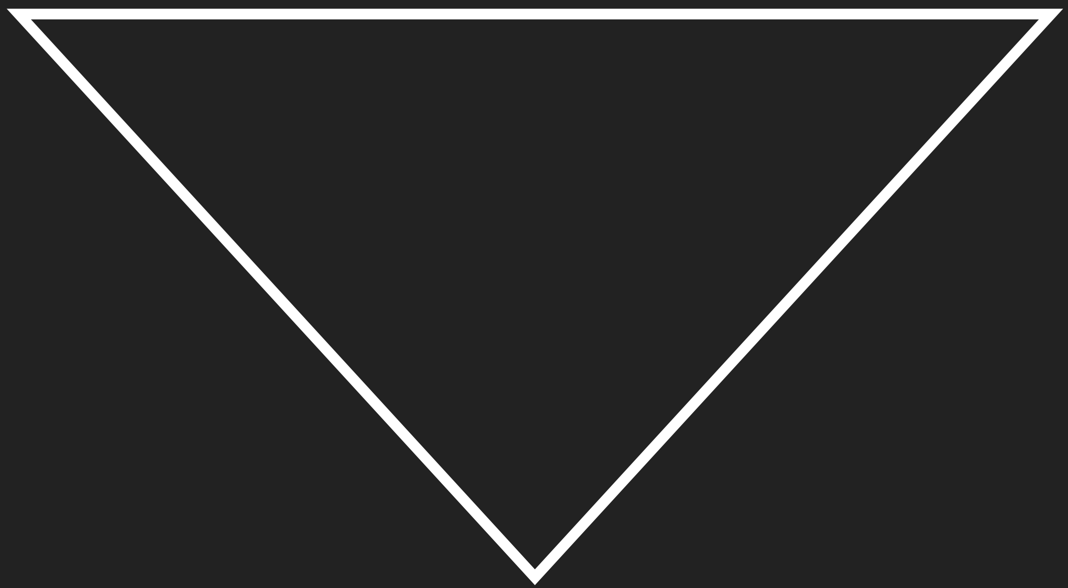
A value

the natural number 1

Meta language term
that represents it

S Z

(in meta language)



Object language term
that represents it

`λf.λx.f (λf.λx.x)`

Generalizing Scott encoding

$$\ulcorner \texttt{Ctor e\_1 ... e\_n} \urcorner$$

(in meta language)

=

$$\lambda\ c_1.\ \lambda\ c_2.\ \ldots\ \lambda\ c_m.\ c_i\ \lceil e_1 \rceil\ \ldots\ \lceil e_n \rceil$$

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

Implementation of Scott encoding from Data

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

  reify 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
  reflect v
    ...

  reify e
      | getTypeRep @a == getTypeRep @Int = unsafeCoerce (reify @Int e)
      | getTypeRep @a == getTypeRep @String = unsafeCoerce <$> (reify @String e)
      | otherwise =
    1 case collectAbs e of -- dissect the nested lambdas
        ([], _) -> Nothing
        (args, body) ->
        2 case spineView body of -- dissect the nested application
            (Var c, rest) -> do
              ctors <- getConstrs @a
              ctor <- lookup c (zip args ctors)
              evalStateT (fromConstrM reifyArg ctor) rest 4
            _ -> Nothing
    where
      reifyArg :: forall d. Data d => StateT [Exp] Maybe d
      reifyArg = do e <- gets head
                    modify tail
                    lift (reify @d e) 3
```

(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

Tying the knot

Now we have a way to take (pretty much) any Haskell value to its representation in Exp.

This can be either a natural number, a color, or … Exp itself.

```
data Exp =
    Var String              x
  | App Exp Exp             e1 e2
  | Abs String Exp          λ x. e
  | StrLit String           "hello"
  | IntLit Int              3
  | MkUnit                  ( )
  deriving (Show, Eq, Data, Typeable)
```

Tying the knot

```
λ> reflect Red
Abs "c0" (Abs "c1" (Abs "c2" (Var "c0")))

λ> reflect (S Z)
Abs "c0" (Abs "c1" (App (Var "c0") (Abs "c0" (Abs "c1" (Var "c1")))))

λ> reflect MkUnit
Abs "c0" (Abs "c1" (Abs "c2" (Abs "c3" (Abs "c4" (Abs "c5" (Var "c5"))))))

λ> reflect (reflect Z)
Abs "c0" (Abs "c1" (Abs "c2" (Abs "c3" (Abs "c4" (Abs "c5" (App (App (Var
"c2") (StrLit "c0")) (Abs "c0" (Abs "c1" (Abs "c2" (Abs "c3" (Abs
"c4" (Abs "c5" (App (App (Var "c2") (StrLit "c1")) (Abs "c0" (Abs
"c1" (Abs "c2" (Abs "c3" (Abs "c4" (Abs "c5" (App (Var "c0") (StrLit
"c1"))))))))))))))))))))))
```

Tying the knot

```
data Exp =
    Var String                  x
  | App Exp Exp                 e1 e2
  | Abs String Exp              λ x. e
  | StrLit String               "hello"
  | IntLit Int                  3
  | MkUnit                      ( )
  | Quasiquote Exp              `(e)
  | Antiquote Exp               ~(e)
  deriving (Show, Eq, Data, Typeable)
```

Tying the knot

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

(no error handling here)

"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, dissertation, 2006

Tying the knot

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

identity function

quoting unit

antiquoting the function application

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?