Reification in Haskell

Bernard Pope
Lee Naish
The University of Melbourne,
Australia

Motivation

How do you represent first-order data generated during the execution of a Haskell program?

Two conflicting requirements:

- 1. close to the syntactic view of the data
- 2. can capture computational effects of cyclic sharing and partial evaluation

Motivation (cont'd)

This work has been driven by the implementation of a (fairly) portable declarative debugger for Haskell.

The debugger must be able to provide *mean-ingful* printable representations of values generated by a computation.

The representation must respect the potential partial evaluation of values.

We would also like to represent cycles in the underlying representation of a value directly.

A simple meta-representation

A simple representation of applications of constructors to zero or more arguments:

```
data Meta = Apply String [Meta]
```

To simplify the presentation of nullary constructors:

```
nullCon x = Apply x []
```

A simple meta-representation (cont'd)

```
True
nullCon "True"

Just False
Apply "Just" [nullCon "False"]

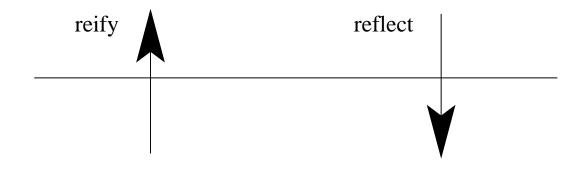
(False, 3.5::Float)
Apply "," [nullCon "False", nullCon "3.5"]

[()]
Apply ":" [nullCon "()", nullCon "[]"]
```

Reification

Reification is the materialization of an abstract concept.

meta level (concrete)



object level (abstract)

We would like the following to hold:

$$reflect \circ reify = identity_{obj}$$
 $reify \circ reflect = identity_{meta}$

Reification (cont'd)

```
class Reify a where
  reify :: a -> Meta
```

An example instance declaration for the [a] type:

Partial evaluation

Evaluation of the meta-representation causes evaluation of the underlying object-level value.

Which gives us the property:

$$reify \perp = \perp$$

We may want to represent the state of evaluation of an object-level value in the metarepresentation.

This requires a version of reify that does not cause its argument to be further evaluated.

We add the constructor Uneval to the Meta type to represent unevaluated components.

Partial evaluation (cont'd)

Consider the program:

```
main = do let list = map id [1..]
    let len = length (take 3 list)
    print len
```

After len has been printed, we represent the state of evaluation for list with:

Partial evaluation (cont'd)

We introduce a new (unsafe) primitive into the language:

```
whnf :: a -> Bool
```

An example instance of the Reify class for the type [a]:

```
instance (Reify a) => Reify [a] where
  reify x
  = if whnf x then
     case x of
     [] -> nullCon "[]"
        (y:ys) -> Apply ":" [reify y, reify ys]
     else Uneval
```

Cyclic sharing

Non-strict programming languages allow the recursive definition of data values:

ones =
$$1$$
 : ones

Under the *lazy* evaluation semantics of Launchbury, *ones* would be represented as:

Detecting cycles

Detecting cycles in a data structure in Haskell is difficult.

- no object identity in Haskell
- pointer equality
- marking nodes

Representing cycles

The representation of cycles is still a topic of further consideration.

Some possible examples:

```
Apply ":" [nullCon "1", Cycle]
```

let x = Apply ":" [nullCon "1", Cycle x] in x

Uses: printing

For the partially evaluated list earlier:

```
(: _ (: _ (: _ _)))
```

Uses: re-evaluation

```
class ReEval a where
  reEval :: a -> Meta -> ()
```

An example instance declaration for the [a] type:

Reflection

The converse operation of reification.

Problem: how to we reflect the meta-values of Uneval and Cycle?

reflect Uneval = undefined

This is not satisfactory.

Solution: embed the object level value in the meta-representation.

First we modify our meta-representation (ignoring cycles for the moment):

We modify the Reify class as follows:

```
class Reify a where
  reify :: a -> Meta a
```

And introduce the Reflect class:

```
class Reflect a where
   reflect :: Meta a -> a
```

An instance of the type a for the Reify class:

```
instance Reify a => Reify [a] where
  reify x
  = if whnf x then
     case x of
     [] -> nullCon "[]"
          (y:ys) -> Apply ":" [Arg y, Arg ys]
     else Uneval x
```

An instance of the type [a] for the Reflect class:

such that we have the unsafe primitive:

```
coerce :: a -> b
```

Example (cont'd)

With reify and reflect we can write programs like:

which evaluates to:

```
3 (: _ (: _ (: _ _)))
True
```

Related work

- Meta-ML
- HOOD
- Dornan's thesis
- generic programming

End notes

- 1. Implementation status?
- 2. Functions?
- 3. Other uses of the meta-representation?
- 4. Dynamic types?