Named and Typed Homoiconicity

Sean Williams, PhD

http://rio-lang.org

Metaprogramming

In Haskell, write a function that, given a tuple of arguments that implement a common typeclass, maps a function in that typeclass over the tuple

Metaprogramming

```
mapT (a, b) f = (f a, f b)

> mapT (1,1) (+1)
(2,2)
```

Metaprogramming

```
mapT (a, b) f = (f a, f b)

> mapT (1::Int,1::Double) (+1)
Couldn't match expected type ‘Int’
  with actual type ‘Double’
(...)
```

Metaprogramming

```haskell
{-# LANGUAGE RankNTypes #-}
mapT :: (Num a, Num b) => (a, b) ->
  (forall x. Num x => x -> x) ->
  (a, b)
mapT (a, b) f = (f a, f b)

> mapT (1::Int,1::Double) (+1)
(2,2.0)
```

Metaprogramming

Now, generalize over tuple arity and typeclass

Metaprogramming

Now, generalize over tuple arity and typeclass

Tuple arity:

- Functional dependencies
- Heterogeneous lists

Metaprogramming

Now, generalize over tuple arity and typeclass

Tuple arity:

- Functional dependencies
- Heterogeneous lists

Typeclass:

- . . . Dragons here

Metaprogramming

This works with no fuss:

```
> ((+1) (1::Int), (+1) (1::Double))
(2,2.0)
```

Doesn't seem very elegant. . .

Metaprogramming

Template Haskell lets us generalize that

We should understand:

- Quasiquote: converts code to formal representation
- Splice: converts formal representation to code

Metaprogramming

```
> runQ [| ((+1) 1, (+1) 1) |]
TupE [AppE
  (InfixE Nothing
  (VarE GHC.Num.+)
  (Just (LitE (IntegerL 1))))
  (LitE (IntegerL 1)),
  (...)
```

Metaprogramming

```
mapT :: Int -> Q Exp -> Q Exp
mapT n f =
  f >>= \f ->
  replicateM n (newName "x") >>= \xs ->
  return (LamE [TupP (map VarP xs)]
    (TupE (map (AppE f . VarE) xs)))

> $(mapT 2 [|(+1)|]) (1::Int,2::Double)
(2,3.0)

> $(mapT 3 [|show|]) (1,'a',"hello")
("1","'a'","\"hello\"")
```

Metaprogramming

Metaprogramming means code generation

This includes type inference and expression generation

Metaprogramming

Peano numerals:

$\mathsf{Z} \equiv 0$

$\mathsf{S\ n} \equiv \mathsf{n} + 1$

$\mathsf{S\ S\ S\ Z} \equiv 3$

Metaprogramming

Haskell

```haskell
{-# LANGUAGE FlexibleContexts, FlexibleInstances,
      FunctionalDependencies, UndecidableInstances #-}

data Z
data S n

class Add a b r | a b -> r, a r -> b
instance Add Z m m
instance Add n m r => Add (S n) m (S r)

add :: Add n m r => n -> m -> r
add _ _ = undefined

> :t add (undefined::S Z) (undefined::S (S Z))
add (undefined::S Z) (undefined::S (S Z)) :: S (S (S Z))
```

Prolog

```prolog
add(0, M, M).
add(s(N), M, s(K)) :- add(N, M, K).
```

Metaprogramming

Type Level

- Discipline
- Premade abstractions
- Paradigm lock
- Tricky interop

Metaprogramming

Type Level

- Discipline
- Premade abstractions
- Paradigm lock
- Tricky interop

Expression Level

- Flexible
- User-made abstractions
- Lots of rope
- Can be portable

Metaprogramming

Why not both? Well. . .

Lisp

Lisp formal language contains three elements:

- Symbols
- Literals
- S-expressions

Lisp

Lists are quasiquoted S-expressions

car and cdr are both programming and metaprogramming constructs

Lisp

The true elegance of Lisp is homoiconicity:

- Syntax is S-expressions
- Formal language is S-expressions

Lisp programming and metaprogramming is S-expression manipulation

Lisp

Lisp is great, but:

Lisp

Lisp is great, but:

- Interpreted

Lisp

Lisp is great, but:

- Interpreted
- Dynamically typed

Lisp

Lisp is great, but:

- Interpreted
- Dynamically typed
- Hetero-list-based, so offset-based

Rio

. . . So we're making a programming language

How do we make name-based homoiconicity?

Rio

How do we make name-based homoiconicity?

- Records instead of lists
  - JSON syntax
  - Considered, not ideal

Rio

How do we make name-based homoiconicity?

- Records instead of lists
    - JSON syntax
    - Considered, not ideal
- Hybrid dictionary-stack
    - A bit weird
    - Surprisingly elegant

Rio

Stack-based?

- Good support for multiple returns
- Composition of multiple returns
- Composition of named returns
- No parentheses

```
8 3 divMod +  =>  4
```

Rio

How do we make statically-typed homoiconicity?

Treat the compiler as single-pass interpreter:

- Statically type literals
- No pointers, just first-class names
- Aggressive inlining, no recursion
- Mangle-by-macro

Rio

Core language provides:

```
({condition} {body})+ {body}? if

"hello "
  { false } { "world" }
  { true } { "place" }
  { "thing" } if ++
    =>  "hello place"
```

Rio

Let's make:

```
val (case {body})+ {body}? switch

({val case =} {body})+ {body}? if
```

Rio

- generate unique name val
- if second item on stack is a block:
  - move top of stack to scratch
- while there's a block at top of stack:
  - move top of stack to scratch
  - fuse top of stack and val onto $\{ = \}$
  - move top of stack to scratch
- bind type at top of stack to ty
- move all blocks from scratch to stack
- push literal of type ty
- push symbols: if bind 'val

Conclusion

Rio metaprogramming is about understanding compiler dynamics

Stack-based promotes point-free style

Homoiconicity lowers the metaprogramming barrier-to-entry

Conclusion

Rio metaprogramming is about understanding compiler dynamics

Stack-based promotes point-free style

Homoiconicity lowers the metaprogramming barrier-to-entry

Is it functional? Who knows

Conclusion

http://rio-lang.org

sjw@imap.cc