Monads

Steve Atkin

Preliminaries

- Developed 1987
- Haskell 98
- Hindley-Milner type system
- Purely functional

Pure (Vs.) Impure

- Pure
  - Haskell, Miranda
    - No side effects
    - Nonstrict – all arguments need not be defined
- Impure
  - ML, Scheme
    - Strict – all arguments must be defined

Pure

- Explicit data flow
  - Value of expression based only on free variabl
    - Referential transparency
  - Computation order irrelevant
    - Lazy evaluation

Types

- Polymorphic

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

User Defined Types

```
data Point a = Pt a a
Pt 2.0 3.0 :: Point Float
data Tree a = Leaf a | Branch (Tree a) (Tree a
type String = [Char]
type Name = String
```

Functions

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- Curried
- add e1 e2 = (add e1) e2
- Applying add to e1 yields a new function whic is then applied to e2

Functions

```
add :: (Integer, Integer) -> Integer
add (x, y) = x + y
```

- Uncurried
- Tuple

Lambda Abstractions

```
add = \x y -> x + y
inc = \x -> x + 1
```

Infix function composition

```
(.) :: (b->c)->(a->b)->(a->c)
F . G = \x -> f (g x)
```

Infix Operators

- Functions
  - Symbols
  - Partial application (section)

(+) :: a->a->a

  - (x+) = \y -> x + y
  - (+y) = \x -> x + y
  - (+) = \x y -> x+y

Infix Operators

- Coerce:
- Infix operator into functional value

  inc = (+ 1)

  add = (+)
- Functional value into infix operator

  x `add` y

  add x y

Layout

- Layout – shorthand for an explicit groupin

```
let { y = a*b                    let y = a*b
    ; f x = (x+y)/y    ==            f x = (x+y)/y
    }                                in f c + f d
in f c + f d
```

Problems

- Blessing and a curse
  - Ultimate in modularity
    - Data in and data out manifest and accessible
    - Maximum flexibility
  - Obscurity
    - Algorithms buried in plumbing

Exceptions

- To add error handling:
  - Pure.
    - Modify each call to check for and handle errors.
  - Impure.
    - Use exceptions, no code restructuring required.

Output

- To add an execution trace:
  - Pure.
    - Modify each call to pass around traces.
  - Impure.
    - Use output as a side effect, no restructuring.

State

- To keep a count of operations:
  - Pure.
    - Modify each call to pass counts around.
  - Impure.
    - Increment a global variable, no restructuring.

References

- A Gentle Introduction to Haskell 98
  - Paul Hudak, John Peterson, and Joseph Fasel
- Monads for Functional Programming
  - Philip Wadler
- Monads for the Working Haskell Programmer
  - Theodore Norvell

References

- Systematic Design of Monads
  - John Hughes and Magnus Carlsson
- Imperative Functional Programming
  - Simon Jones and Philip Wadler
- What the Hell are Monads
  - Noel Winstanley