Language Oriented Programming

Iavor S. Diatchki

February 2019

Thesis

1. It is convenient to implement the sub-components of a system in *custom programming languages*, tailored to the needs of the component.
2. This is the essence of "monadic" programming.
3. What features does a host languge need to support this style of software development?

Language Primitives

Expressions are pure:

- evaluate to values
- flexible evalutaion order
- example: combinatorial circuits

Statements are effectful:

- have a notion of sequencing
- do this, then do that
- example: a recipie

Monads

A *monad* is a language that uses statements.

Notation

```
s : L t
```

- `s` is a statement,
- in language `L`
- which produces a value of type `t`.

Example:

```
getchar() : C int
```

Sequencing Statements

Combine statements to form more complex ones:

If:

- `s1 : L a`
- `s2 : L b`, with a free variable `x : a`

Then:

```
do { x <- s1; s2 } : L b
```

Promoting Expressions to Statements

If:

```
e : a          -- `e` is an expression
```

Then:

```
pure e : L a
```

In many languages this is implicit.

Monad Laws = Resonable Behavior

The grouping of staments is not important:

```
do { y <- do { x <- s1; s2 }; s3 } =
do { x <- s1; do { y <- s2; s3 } } =          -- modulo nami
do { x <- s1; y <- s2; s3 }
```

Expression statements don't have effects:

```
do { x <- pure x; s } =
s                     =
do { x <- s; pure x }
```

Effects

- Monadic strucutre = bare minimum.
- We need statemtns that do something.

Example:

```
getGreeting :: IO String
getGreeting =
  do putStrLn "What is your name?"
     x <- getLine
     pure ("Hello, " ++ x)

main :: IO ()
main =
  do msg <- getGreeting
     putStrLn msg
```

Classes of Effects

- Data effects (aka "variables")
  - Read-only variables (e.g., configuration)
  - Mutable variables
  - Write-only variables (aka "collectors", e.g., logs)
- Control effects
  - Exceptions (early termination)
  - Backtracking (search)
  - Continutations (coroutines)