

# Language Oriented Programming

Iavor S. Diatchki, Galois Inc.

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 language need to support this style of software development?

# Language Primitives

*Expressions* are pure:

- ▶ evaluate to values
- ▶ flexible evaluation order
- ▶ example: combinatorial circuits

*Statements* are effectful:

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

# 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:

$\text{do } \{ x \leftarrow s1; s2 \} :: L\ b$

# Promoting Expressions to Statements

If:

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

Then:

`pure e :: L a`

In many languages this is implicit.

# Monad Laws = Reasonable Behavior

The grouping of statements is not important:

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

*-- modulo naming*

Expression statements don't have effects:

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



# Effects

- ▶ Monadic structure = bare minimum.
- ▶ We need statements 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
```

# Three Questions

1. How do we specify the features of a language?
2. How do we write programs in a language?
3. How do we execute programs in the language?

# Modular Language Construction

Start with a language of *primitives*, and extended with desired *features*.

```
type MyPL =  
  DeclareLanguage  
    [ F3      -- Feature 3  
    , F2      -- Feature 2  
    , F1      -- Feature 1  
    ] Prim    -- Language of primitives
```

Primitive language examples:

- ▶ IO: a language for interacting with the OS
- ▶ Pure: no primitive language

# Common Features

## Data effects (aka variables)

- ▶ `Val x t` adds an immutable variable
- ▶ `Mut x t` adds a mutable variable
- ▶ `Collector x t` adds a collector variable

## Control effects

- ▶ `Throws t` add support for exceptions
- ▶ `Backtracks` add support for backtracking

# Feature Dependencies

The order in which features are added to a language is important (sometimes):

- ▶ Data effects are *orthogonal*: order is not important.
- ▶ Control effects are not: order in feature list matters.

Rule:

*Existing features take precedence over new features.*

## Example

```
type PL1 =  
  DeclareLanguage  
    [ Throws e  
    , MutVar x t  
    ] Pure
```

```
type PL2 =  
  DeclareLanguage  
    [ MutVar x t  
    , Throws t  
    ] Pure
```

How do exceptions affect changes to x?

- ▶ PL1: changes survive exceptions
- ▶ PL2: changes are rolled back on exception

# Writing Programs

- ▶ Need a common notation for similar features across multiple language (e.g. read a variable).
- ▶ Exact behavior is determined by the language.

<code>readVal</code>	<code>:: HasVal x t m</code>	<code>=&gt; x -&gt; m t</code>
<code>getMut</code>	<code>:: HasMut x t m</code>	<code>=&gt; x -&gt; m t</code>
<code>setMut</code>	<code>:: HasMut x t m</code>	<code>=&gt; x -&gt; t -&gt; m ()</code>
<code>appendTo</code>	<code>:: HasCollector x t m</code>	<code>=&gt; x -&gt; t -&gt; m ()</code>
<code>throw</code>	<code>:: Throws t m</code>	<code>=&gt; t -&gt; m a</code>
<code>backtrack</code>	<code>:: Backtracks m</code>	<code>=&gt; m a</code>
<code>orElse</code>	<code>:: Backtracks m</code>	<code>=&gt; m a -&gt; m a -&gt; m a</code>

# Running Programs

Each feature can be “compiled” away:

```
val      :: Language m => (x := t) -> Val x t m a -> m a
```

```
mut      :: Language m => (x := t) -> Mut x t m a -> m a
```

```
collector :: Language m => Col x t m a -> m (a, [t])
```

```
throws   :: Language m => Throws t m a -> m (Except t a)
```

```
backtracks :: Language m => Maybe Int -> Backtracks m a -> m a
```

Or, we can compile and run the whole program:

```
run :: Run m => m a -> ExeResult m a
```



## Scoped Statements

Allow for “nested” statement execution.

```
letVal      :: LetVal x t m      => x -> t -> m a -> m a
collect     :: CanCollect x t m => x -> m a -> m (a, [t])
try         :: CanCatch t m      => m a -> m (Except t a)
findUpTo    :: CanSearch m       => Maybe Int -> m a -> m [a]
```

Quite useful, much trickier semantics.

# Bigger Example

A language for a type-checker:

```
type TCLang =  
  [ Throws TCErrors -- Critical errors  
    , Val Env      (Map Name Type) -- Types of free variables  
    , Mut Subst    (Map TVar Type) -- Inferred types  
    , Col Ctrs     (Set Ctr)       -- Collected constraints  
    , Col Warns    (Set Warn)      -- Warnings  
  ] IO -- Interact with solvers
```

# Haskell as a Host Language

Haskell is a *great* language for experimenting with language design:

- ▶ Type system tracks language fragments
- ▶ Lazyness for custom control flow operators
- ▶ Functions for binders and modelling jumps
- ▶ Overloading for reusable notation

# Drawbacks of Embedding in Haskell

- ▶ Performance can be difficult to reason about
- ▶ Embedded notation not as neat as custom syntax
- ▶ Potentially confusing type errors
  - ▶ although custom type errors do help

# Idea

Experience with Haskell has identified a set of useful abstractions.

*Could we design a language that supports this style of programming directly?*