関数型言語を作ろう

Let's Make a Functional Language!

RubyKaigi2015

NaCl

yhara (Yutaka Hara)

Or:

Rubyistのための型推論入門

Type Inference 101 for Rubyist

RubyKaigi2015

NaCl

yhara (Yutaka Hara)

Agenda

1. What is "Type Inference?"
2. Hindley-Milner type system
3. Implementation

- https://github.com/yhara/rk2015orescript

OreScript

```
f = fn(x){ printn(x) }
f(2)   //→ 2
```

Difference from JavaScript

```
f = fn(x){ printn(x) }
f(2)   //→ 2
```

- No semicolon
- s/function/fn/

Myself

- @yhara (Yutaka Hara)
- NaCl (Matsue, Shimane)
- Making software with Ruby

My blog[1]

Route 477

2015-09-14

■ [ruby] gemを作るときに実行ファイルをbin/以下ではなくexe/以下に置くことについて

例えばあなたがコマンドラインからピザを注文するようなRubyGemsを書いているとすると、今まではbin/order_pizzaに実行用のファイルを置いていたと思うけど、それがexe/order_pizzaに変わるかもしれない、という話。

仕様変更なの？

実行ファイルをどこに置くかというのは完全に慣習の話で、.gemspecの仕様上は好きなところに置けるようになっている。なので既存のgemが動かなくなるという話ではない。
ただbundle gemコマンドでgemを新規生成したときに、以前のBundlerでは.gemspecが以下のようになっていたのに対し、

```
spec.bindir        = "bin"
spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
```

Bundler 1.8からはこの箇所が以下のようになる。

```
spec.bindir        = "exe"
spec.executables   = spec.files.grep(%r{^exe/}) { |f| File.basename(f) }
```

そのため今から新しくbundle gemしてgemを作るときは、実行ファイルはexe/以下に置く(か、.gemspecの当該箇所を編集する)必要がある。

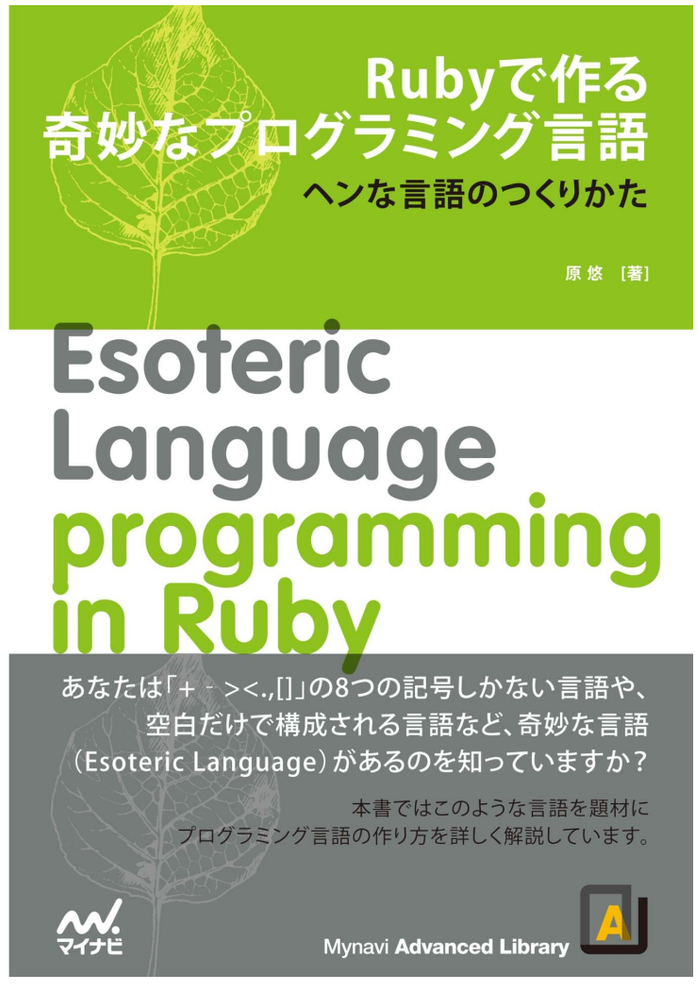
なんで変わったの？

こっちには「うっかり開発用スクリプトをbin/に入れてしまうと、gem install時にインストールされてしまってうざい」みたいなこと

[1] http://route477.net/d/

Me and Ruby

- Enumerable#lazy (Ruby 2.0~)
  - Note: I'm not a Ruby committer
- TRICK judge
- 『Rubyで作る奇妙なプログラミング言語』(Making Esoteric Language with Ruby)



1. What is "Type Inference"?

What is "Type"?

- Type = Group of values
  - 1,2,3, ... => Integer
  - "a", "b", "c", ... => String

What is "Type"?

- Ruby has type, too! (Integer, String,...)
- Ruby variables do not have type, though

```
a = 1
a = "str"   # ok
```

- This is error in C

```
int a = 1;
a = "str";  // compile error!
```

Pros of static typing

1. Optimization

2. Type check

```
def foo(user)
  print user.name
end

foo(123)
#=> NoMethodError: undefined method `name' for 123:Fixnum
```

Cons of static typing

- Type annotation?

```
Array<Integer> ary = [1,2,3]
ary.map{|Integer x|
    x.to_s
}
```

Type inference

```
-- Haskell
ary = [1,2,3]
map (\x -> show x) ary
```

- No type annotations here

RECAP

- Type = Group of values
  - Static typing
    - Check type errors
    - Optimization
  - Don't want to write type annotations
    => Type Inference

Various "Type Inference"

- C#:
  - `var ary = [1,2,3];`
- Haskell, OCaml:
  - Can omit type of function arguments, etc.
  - Hindley-Milner type system

2. Hindley-Milner type system

What is "type system"?

- System of types, of course :-)
- Set of rules about type
  - Decides which type an expression will have
  - Decides which types are compatible
  - eg. Inheritance
- Every language has its own type system

What is "type system"?

- Hindley-Milner type system
  - Haskell = HM + type class + ...
  - OCaml = HM + variant + ...
  - OreScript = HM (slightly modified)
- Has an algorithm to reconstruct types
  - without any type annotation(!)

OreScript language spec

- Literal
  - eg. `99`, `true`, `false`
- Anonymous function
  - eg. `fn(x){ x }`
- Variable definition
  - eg. `x = 1`
  - eg. `f = fn(x){ x }`
  - (Note: You can't reassign variables)
- Function call
  - eg. `f(3)`

Only unary function is supported

- Don't worry, you can emulate binary function

```
f = fn(x, y){ ... }
f(1, 2)
```

↓

```
f = fn(x){ fn(y){ ... } }
f(1)(2)
```

Type system of OreScript

- <type> is any one of ...
  - Bool
  - Number
  - <type> → <type>
    - eg. `is_odd :: Number → Bool`
- Checks
  - Type of a and x must be the same

```
f = fn(a){ ... }
f(x)
```

Type inference of OreScript

- Given
  - `f = fn(x){ is_odd(x) }`
  - is_odd :: Number → Bool
- step1 Assumption
  - f :: (1) → (2)
    - x :: (3)
- step2 Equations
  - (1) == (3), (3) == Number, (2) == Bool
- step3 Resolve
  - (1) == Number, (2) == Bool, (3) == Number
  - f :: Number → Bool

RECAP

- Type system = set of rules on types
- Hindley-Minler type system
  - Reconstruct types without annotation
    - Assume, Build equations, Resolve

3. Implementation of OreScript

bin/ore_script

```
$ cat a.ore
printn(123)
$ ./bin/ore_script a.ore
123
```

bin/ore_script

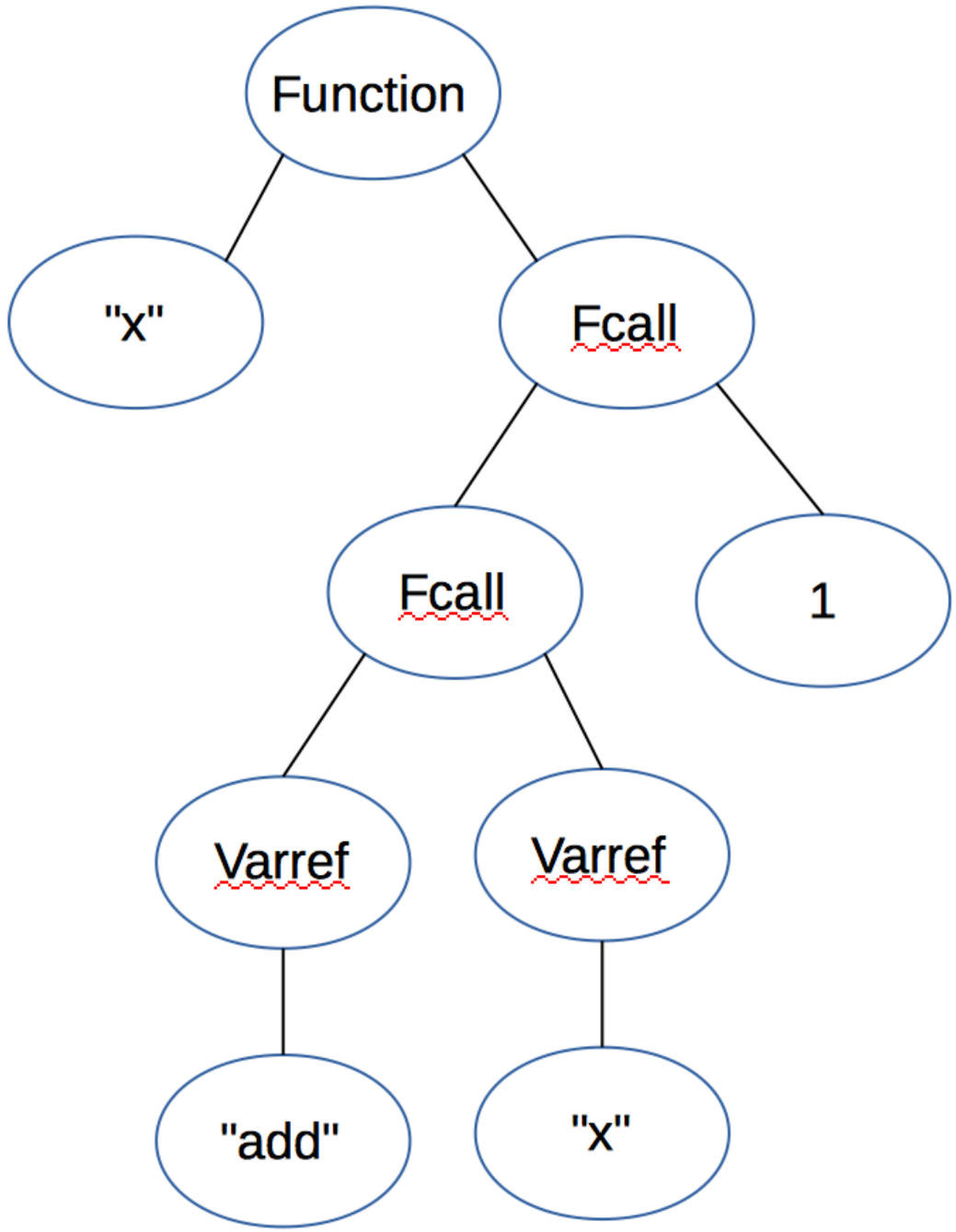
```
#!/usr/bin/env ruby
require 'ore_script'

# 1. Parse
tree = OreScript::Parser.new.parse(ARGF.read)
# 2. Type check
OreScript::TypeCheck.new.check(tree)
# 3. Execute
OreScript::Evaluator.new.eval(tree)
```

1. OreScript::Parser

- Convert source code into a tree (parse tree)

```
fn(x){ add(x)(1) }
```



Parser library for Ruby

- racc gem
- treetop/parslet/citrus gem
- Write by hand
  - Recursive Descent Parsing
  - eg. https://github.com/yhara/esolang-book-sources/blob/master/bolic/bolic.rb

racc gem

parser.ry:

```
    expression : let
               | function
               | fcall
               | if
               | varref
               | literal
               | '(' expression ')'
    let : VAR '=' expression
    function : 'fn' '(' params ')' '{' expressions '}'
    fcall : expression '(' args ')'
    if : 'if' '(' expression ')' '{' expressions '}'
         'else' '{' expressions '}'
...
```

Result of parsing

```
fn(x){ add(x)(1) }
```

```
[:exprs,
 [[:function,
   "x",
   [:exprs,
    [[:fcall,
      [:fcall, [:varref, "add"], [:varref, "x"]],
      [:literal, "Number", 1.0]]]]]]]
```

bin/ore_script

```
#!/usr/bin/env ruby
require 'ore_script'

# 1. Parse
tree = OreScript::Parser.new.parse(ARGF.read)
p tree
```

3. OreScript::Evaluator

- Walk the tree and do what is expected
  `[:if, cond_expr, then_exprs, else_exprs]`

```
def eval_if(env, cond_expr, then_exprs, else_exprs)
  cond = eval_expression(env, cond_expr)
  case cond
  when Value::TRUE
    eval_expressions(env, then_exprs)
  when Value::FALSE
    eval_expressions(env, else_exprs)
  else
    raise "must not happen"
  end
end
```

bin/ore_script

```
#!/usr/bin/env ruby
require 'ore_script'

# 1. Parse
tree = OreScript::Parser.new.parse(ARGF.read)
# 3. Execute
OreScript::Evaluator.new.eval(tree)
```

What happens if ...

```
f = fn(x){ add(x, 1) }
f(true)   // !?
```

- Where's type inference?
- Why we wanted type inference
  - "Want to **check types** without type anottations"

2. OreScript::TypeCheck

```
#!/usr/bin/env ruby
require 'ore_script'

# 1. Parse
tree = OreScript::Parser.new.parse(ARGF.read)
# 2. Type Check (Type Inference here)
OreScript::TypeCheck.new.check(tree)
# 3. Execute
OreScript::Evaluator.new.eval(tree)
```

Type Inference = Type Check

```
f = fn(x){ is_odd(x) }
f(true)   // !?
```

- `f :: (1) → (2)`
  `x :: (1)`
  `is_odd :: Number → Bool`
- Bool == (1)
  (1) == Number
  (2) == Bool
- ∴ Bool == Number // !?

Type Inference = Type Check

- Infer type before executing program
- If program has an error:
  - Bool == Number (unsatisfiable)
- Otherwise:
  - The program has consistent types
    (No contradiction detected)

RECAP

- bin/ore_script
  - 1. Parse
  - 2. Type check (= Type inference)
  - 3. Execute

Implementation of type inference

Three classes for type

- Type::TyRaw
  - A type already known (Number, Bool, etc.)
  - `99 :: #<TyRaw "Number">`
- Type::TyFun
  - Function type
  - `f :: #<TyFun #<TyRaw "Number"> -> #<TyRaw "Bool">>`
- Type::TyVar
  - A type not yet known
  - `x :: #<TyVar (1)>`
  - `f :: #<TyVar (2)>`

Three steps (recap)

1. Assume types
2. Extract type equations
3. Resolve equations

Actual steps

- 1. Assume types
  - 2. Extract type equations
    - 3. Resolve equations
      - 2. Extract type equations
      - 3. Resolve equations
- ...

OreScript::TypeCheck#infer

```
    def infer(env, node)
      ...
    end

tree = Parser.new.parse("99")
infer(..., tree)
#=> [...>, Ty(Number)]

tree = Parser.new.parse("f = fn(x){ add(x)(1) }"))
infer(..., tree)
#=> [..., Ty(Number -> Number)]
```

OreScript::TypeCheck#infer

```
def infer(env, node)
  ...
  when :fcall
    ...
    result_type = TyVar.new
    s1, func_type = infer(env, func_expr)
    s2, arg_type = infer(env.substitute(s1), arg_expr)
    ...
    equation = Equation.new(
      func_type,
      TyFun.new(arg_type, result_type)
    )
    ...
end
```

TypeCheck.unify(*equations)

- Pop one from `equations`
  - `#<TyFun ty1 -> ty2>` == `#<TyFun ty3 -> ty4>`
    - ty1 == ty3
    - ty2 == ty4
  - `#<TyRaw "Number">` == `#<TyRaw "Number">`
    - just ignore
  - `#<TyVar (1)>` == `#<TyRaw "Number">`
    - Add `(1) == "Number"` to the answers
    - Replace `(1)` with `#<TyRaw "Number">` in rest of the `equations`
- Repeat until all equations are removed

Further topics

Downside of static type check

- May reject "valid" program

```
id = fn(x){ x }
id(99)      //→ 99
id(true)    //→ true??
```

let

```
let id = fn(x){ x } in
  id(99)
  id(true)
```

let-poly branch[2]

```
id = fn(x){ x }    // id :: ∀(1). (1) → (1)
id(99)
id(true)
```

[2] https://github.com/yhara/rk2015orescript/tree/let-poly

let-poly branch[2]

```
id = fn(x){ x }     // id :: ∀(1). (1) → (1)
id(99)              // ←id here :: (2) → (2)
id(true)            // ←id here :: (3) → (3)
```

[2] https://github.com/yhara/rk2015orescript/tree/let-poly

Acknowledgements

- 『Types And Programming Language』(TAPL)
  - Japanese edition: 『型システム入門』
- 『プログラミング言語の基礎概念』
  - see also(PDF)
- 『プログラミング言語の基礎理論』(絶版)
- 『アルゴリズムW入門』 (同人誌)
- 『Scala By Example』chapter16
- Ibis (Type infrence written in JavaScript)

Summary

- OreScript
  - Small language with type inference
  - Type check without type annotation
- Type inference (= type check)
  - Build type equeations
  - Resolve type equeations
- https://github.com/yhara/rk2015orescript