Rogue programming languages specs

Dmitry Kovanikov

Implementation language: Haskell

Target platform: LLVM

Language features

Rogue programming language is mix of imperative and functional paradigms. You can write algorithms step by step. But language has plenty of functional features. It's statically typed with local type inference.
As a general purpose programming language it has commonly used set of basics:

1. Variable declaration.
2. Basic arithmetic, logic and comparison operations.
3. Function declaration and calling.
4. Primitive types: Int, Bool, Double, Unit (Word? String? BigInteger?)

Imperative language features are:

1. Variables can be mutable and immutable.
2. Named function arguments with default values.
3. Control-flow constructs: if-then-else, while and for loops.
4. TODO: Arrays with size in type?

Functional language features are:

1. Pattern matching on constants.
2. Higher-order functions.
3. Anonymous functions.
4. Currying and partial application.
5. Classification of functions on pure and with side-effects.
6. Algebraic immutable data types.
7. TODO: parametric polymorphism.

TODO: fully-compatible with Haskell.

Syntax examples

Listing 1: Variables declaration

```
mut x: Int = 0
mut y = 3
let z = true    // ‘z‘ has type Bool
y = 5
```

Using **mut** keyword you can create mutable variables, and with **let** — immutable. Note, that you can omit type of variable, if it can be inferred from it's local context. But variable initialization is required.

Listing 2: Function declaration

```
pure f : (x: Double) -> (y: Double) -> Double {
    let z = x + y
    return z
}

dirty
g : (b: Bool) -> (mut x: Int) -> (y: Int = 5) -> Int
g (..) = {
    if b { x += y }
    return x
}
```

Arguments of functions separated with $\to$ symbol. They should have names and type. Arguments can have default values. Last type doesn't have name "— it is the type of function result. All functions should be marked either **pure** or **dirty**. If function has side-effects (changes some of it's arguments or some global variables or does some IO) then it should be marked as **dirty**. Function block should have **return** keyword if result type of function is not **Unit**.
Next calls of function `g` are valid:

Listing 3: Function calls

```
g true someX 4
g true someX    // creates function with type ‘Int -> Int‘
g(b = true, x = someX)  // calls ‘g‘ with ‘y = 5‘
g(x = someX)  // creates function with type Bool -> Int
g() // creates function with type ‘Bool -> mut Int -> Int‘
```

So `g {x = someX} true` is valid call whereas `g true {x = someX}` is not because partial function application loses all information about arguments names. Though by convention you shouldn't write `g {x = someX} true` because it's less obvious what this function call does.
TODO: Call like `g(..)`
Functions also can perform pattern matching, have guards, etc.

Listing 4: Pattern matching

```
dirty h : (b: Bool) -> (mut x: Int) -> (y: Int) -> Int
h (y = 0) = {
    x = x - 3
    let a = x / 2
    return x + a
}
h (y = 1)
    | x > 1 = x - 1
    | else  = { let t = x % 2; return t }
h false 1 _ = x + 10  // ‘y‘ is not available here
h true (..) = y + 10  // {..} for keeping rest arguments names
h (..) = x + 1
```

Here's example of function that reads two integers from input and performs binary pow aglorithm.

Listing 5: Binary pow algorithm

```
dirty binPow : Unit {
    mut k = readInt()
    mut n = readInt()
    mut res = 1
    while k > 0 {
        if k % 2 == 1 {
            res *= n
        } else {
            skip // `skip` is empty operator
        }
        n = n * n
        k = k / 2
    }
    print res
}
```

Example of calling with higher-order functions.

Listing 6: Higher order functions

```
dirty decrementAndCheck : (mut x: Int)
                        -> (p: mut Int -> Int -> Bool)
                        -> Bool
{
    x = x - 1
    return (p x 3)
}
```