Radlang programming language

Radosław Rowicki

April 11, 2019

Contents

1 About

Radlang is a pure functional programming language that targets to inherit most of Haskell's features. It supports higher order polymorphic algebraic datatypes, full type inference, typeclasses (known as intefaces) and rich syntax. The evaluation order is lazy by default, however the user is allowed to manage it to some extent.

In the future it will probably support memory management via automated garbage collection and have read-eval-print loop.

2 Syntax

```
kind ::= "Type" | kind "->" kind | "(" kind ")"

general_typename ::= "~" upper_case_string

type ::= general_typename (* type var *)
       | upper_case_string (* rigid type *)
       | type "->" type (* function type *)
       | type {type}+ (* type application *)
       | "(" type ")"

predicate ::= type "is" upper_case_string

qualified_type ::= [predicate {"," predicate}* "|"] type

literal ::= signed number
          | "'" character "'"
          | "\"" {escaped_character}* "\""

pattern ::= lower_case_string (* var name *)
          | "_" (* wildcard *)
          | lower_case_string "@" pattern (* "as" pattern *)
          | literal
          | upper_case_string {pattern}*
          | "(" pattern ")"

typedecl ::= lower_case_string ":" qualified_type

datadef ::= lower_case_string [pattern] ":=" expr

binding ::= datadef | typedecl

for_unit ::= pattern "<-" expr | expr
```

```
expr ::= lower_case_string (* variable *)
       | upper_case_string (* constructor *)
       | literal
       | expr {expr}+ (* application *)
       | "let" binding {"|" assignment}* "in" expr
       | {"if" expr "then" expr}+ "else" expr (* multi way if *)
       | "\ " pattern "->" expr (* lambda *)
       | "match" expr "with" {"|" pattern "->" expr}+ (* pattern match *)
       | "for" "{" [for_unit {"|" for_unit}*] "}" expr (* monadic for comprehension *)
       | "[" [expr {"," expr}*] "]" (* list sugar *)
       | "(" expr ")"

constructor_def ::= upper_case_string {type}*

newtype ::= "newtype" upper_case_string {"(" general_typename ":" kind ")"}* ":=" {cons

interface ::=
  "interface" upper_case_string "(" general_typename ":" kind ")"
    ["implies" upper_case_string {upper_case_string}*] (* superclasses *)
    "{"
     {typedecl ";;"}*
    "}"

impl ::= (* interface implementation *)
  "impl" qual_type "for" upper_case_string
   "{"
    {datadef ";;"}*
   "}"

program ::= {(newtype | typedecl | datadef | interface | impl) ";;"}*
```

3 Overview

3.1 Basic definitions and examples

The program is a set of data definitions, type declarations, newtype definitions, interface declarations and implementations of the interfaces. The infix operators probably won't be supported. Program must contain 'main' definition of any type – it will be the point where the evaluation starts. 'main'

will be deeply forced and will never contain any unevaluated thunk.

3.1.1 Hello world

```
main := "hello world";;
```

3.1.2 Use of toplevel function and `if` statement

```
identity x := x;;

main := if identity True then identity False else False;;
```

3.1.3 Type declaration and pattern matching

```
fun : Int -> Bool;;
fun 0 := True;;
fun x = match x with
  | 4 -> False
  | _ -> eqInt x 7
;;
```

Note that no variable may appear twice in a single set of patterns. Differing numbers of function arguments are not supported, so following code **won't** pass the syntax check:

```
f x := 1;;
f := const 2;;
```

3.1.4 For comprehension

This construction uses implicitly `bind` function like in Haskell's `do` notation:

```
main := for
  { x <- x_m
  | y <- y_m
  | guard (gtInt y x)
  } unit (plusInt x y)
;;
```

3.1.5 Type declarations and data definitions

```
x : Int;;

notEq : ~A is Eq -> ~A -> ~A -> Bool;;

mplus : ~A is Semigroup, ~M is Monad | ~M ~A -> ~M ~A -> ~M ~A;;
```

In opposite to Haskell one may define variable without any value. To do so, the programmer must declare its type only:

```
bot : ~A;;

bot_int : Int;;
```

3.2 New type definition

Types can be defined in casual ADT terms, however the programmer must explicitly provide kind annotation for every type-argument:

```
newtype Bool := True | False;;

newtype List (~A : Type) := Nil | Cons ~A (List ~A);;

newtype StateT (~S : Type) (~M : Type -> Type) (~A : Type) :=
  StateT (~S -> ~M (Pair ~S ~A))
```

Such data may be deeply pattern matched:

```
f l := match l with
  | Nil -> 0
  | Cons 3 (Cons x _) -> 1
  | _ -> 2
```

3.3 Laziness management

Every value is treatet lazily, that means following code

```
bot : ~A;;

main := (\a b -> a) 3 bot;;
```

will successfully return 3. However there are two built in functions that are able to interfere this behavior:

- `force : ~A -> ~B -> ~B` – that forces its first argument to WHNF and returns the second (just like Haskell's `seq`)
- `deepForce : ~A -> ~B -> ~B` – that deeply forces all possible parts of the first argument

`main` function will always implicitly call `deepForce` on its value. Examples:

```
test (Cons _ _) := True;;
test _ := False;;
bot : ~Any;;

main0 := test Nil;; -- False

main1 := test bot;; -- out of domain error

main2 := test (Cons bot bot);; -- True

main3 := force bot True;; -- out of domain error

main5 := let x := Cons bot bot in True;; -- True

main4 := force (Const bot bot) True;; -- True

main6 := deepForce (Cons bot bot) True;; -- out of domain error
```

3.4 Interfaces

Interfaces are technically same thing as typeclasses or traits in other languages. They may inherit each other as long as they do not form cycles (that would be a true deviation).

One of the most fancy features of this system is that implementation may provide definitions for any upper interface in an inheriting interface:

```
interface Semigroup (~S : Type) {
  plus : ~S -> ~S -> ~S;;
};;
```

```
interface Monoid (~S : Type) implies Semigroup {
  null : ~S;;
};;

impl Int for Monoid {
  plus := plusInt;;
  null := 0;;
};;
```

3.5 Stacktrace

Running program will keep two different stacktraces to ease debugging. The stacktraces will appear on every runtime error.

One may wonder what is the reason to have two of them. The answer is simple – laziness. Consider following code:

```
main :=
  let x := f 1
  in g x;;

f x := divInt x 0;; -- division by zero!

g x := deepForce x x;;
```

The program will surely return an error, but what should its stacktrace be? In strict languages it would be obvious – `[main, f, divInt]`, because this is the place where runtime failed. However, because Radlang is lazy the error will be thrown at `[deepForce, divInt]`. So which stacktrace is correct? Being prepared for situation where the user can't decide, I provide both of them – first one is called "definition stacktrace", and the second one is "evaluation stacktrace".

What happened to the `g` function in the evaluation stacktrace? The answer is simple – it is just not present there. `g` was fully evaluated and returned thunk that contains `deepForce x x` with `x` assigned to `f 1`. The error was actually thrown out of the main function so it is not mentioned either!

4 What may be included, but doesn't have to

- infix operators

- explicitly typed GADTs
- garbage collection
- type aliasses
- REPL
- tensorflow bindings

5 MIMUW course grade expectation

I expect to get maximum number of points if I finish all the features declared here (excluding "what may be included" section). Typeclasses are quite complicated in terms of typechecking and semantics, so in my opinion I would deserve it. I am going to take care over the syntax and in the end my target is to provide software that could be used for educational purposes.