English title

(Wymagający złamania wierszy
tytuł pracy w języku polskim)

Konrad Werbliński

Praca licencjacka

Promotor: dr Filip Sieczkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 sierpnia 2019

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

Chapter 2

Language description

2.1 Language features

Bestrafer's syntax was designed to be concise, expressive, readable and beautiful. It was strongly influenced by Haskell, but modified to be indentation-insensitive for greater flexibility in writing beautiful code and ease of parsing.

```
//Single-line comment
/*
  Multi-line comment
*/

def fac :: Int -> Int
def fac 0 = 1
def fac n = n * fac (n - 1)

def ack :: Int -> Int -> Int
def ack m n = case (m, n) of
  | (0, n) -> n + 1
  | (m, 0) -> ack (m - 1) 1
  | (m, n) -> ack (m - 1) (ack m (n - 1))

def main :: ()
def main =
  printInt (fac 5) `seq`
  printInt (ack 3 1)
```

Our language supports Haskell-like top-level pattern matching in definitions. Type annotations for top-level definitions are obligatory due to bidirectionality of the type system. We use call-by-value evaluation strategy like many mainstream functional languages. Program is evaluated from the top to the bottom of the source file (with

a minor subtlety broader described in the chapter 4). All of the definitions on the top-level are mutually recursive. One can also define nested functions using `rec` keyword.

```
def fib :: Int -> Int
def fib n =
  rec :: (Int, Int) -> Int -> Int :
    f lasts n = case n of
      | 0 -> fst lasts
      | n -> f (snd lasts, fst lasts + snd lasts) (n - 1)
  in f (0, 1) n
```

Bestrafer supports all of the typical IO operations including reading and writing files as well as parsing and printing values of primitive types from and to standard input-output. IO operations may be performed at any point in program, following the style of several languages in the ML family. It also supports exception handling with `error` keyword for throwing errors and `try-catch` block for catching user thrown (`RuntimeException`) and builtin (`IOException`, `ArithmeticException`) exceptions. We can use optional variable in exception pattern for extracting the error message.

```
def checkPassword :: String -> ()
def checkPassword s =
  if s == "Rammstein" then
    ()
  else
    error: "Password is incorrect"

def main :: ()
def main =
  try:
    let password = getLine () in
    checkPassword password `seq`
    let x = readLnInt () in
    printInt (1000 / x) `seq`
    let filename = getLine () in
    readFile filename |> putStrLn
  catch:
    | IOException e -> putStrLn e
    | ArithmeticException -> putStrLn "Division by zero"
    | RuntimeException e -> putStrLn e
    | Exception e -> putStrLn e
```

Bestrafer allows user to define his own generalized algebraic data types (GADTs)

using the `data` keyword. There are two kinds of parameters in GADT definition:

- named (denoted with a name starting with `'` followed by a capital letter) which work exactly like parameters of standard algebraic data types in languages like Haskell or OCaml.
- unnamed (denoted with their kind: `*` or `N`) which may be set by the user to any type of the specified kind, thus providing GADT functionality.

```
data Maybe 'A where
  | Nothing :: Maybe 'A
  | Just :: 'A -> Maybe 'A
```

Our language also supports defining data types without value constructors, which may be used as annotations in GADTs, like types `Ok` and `Fail` used to annotate type `Either` in the following example.

```
data Ok
data Fail

data Either * 'A 'B where
  | Left  :: 'A -> Either Fail 'A 'B
  | Right :: 'B -> Either Ok 'A 'B
```

A flagship data type of Bestrafer language is a list indexed by its length, traditionally called `Vec`.

```
data Vec N 'A where
  | []  :: Vec 0 'A
  | (:) :: forall n : N . 'A -> Vec n 'A -> Vec (S n) 'A
```

Using the above definition we can write `map` function, which type encodes the proof that the resulting `Vec` has the same length as the input one.

```
def map :: forall n : N, a : *, b : * .
  (a -> b) ->
  Vec n a ->
  Vec n b
def map _ [] = []
def map f (x : xs) = f x : map f xs
```

To give programmer full flexibility and expressive power our language also has a standard non-indexed `List` data type.

```
data List 'A where
  | {}  :: List 'A
  | (;) :: 'A -> List 'A -> List 'A
```

Bestrafer also supports existential types, but unlike in Haskell and OCaml their usage is not tied to data types declarations. Instead they can be used freely like any other type constructor. The following implementation of a `filter` function (taken from Bestrafer's standard library) utilizes existential type to express the fact that we cannot predict length of the resulting `Vec`. We use `let` expression to unpack result of recursive call from the existential type, thus ensuring that the type variable describing length of `tail` is inserted to the context before the subtyping starts.

```
def filter :: forall n : N, a : * .
  (a -> Bool) ->
  Vec n a ->
  exists k : N . Vec k a
def filter _ [] = []
def filter p (x : xs) =
  let tail = filter p xs in
  if p x then
    x : tail
  else
    tail
```

Quantifiers are always explicit to enforce conscious kind specification and emphasize connection to a type theoretic core. To articulate this connection even more instead of writing `forall`, `exists` and `\x -> x` one can write $\forall$, $\exists$ and λ `x -> x`.

2.2 GADT examples

Matrix algebra

We can make a great use of Bestrafer's indexed Vec type to implement matrix algebra operations. Now the types provide the proof that the matrix operations that we defined produce results with correct dimensions and impose restriction on input arguments which ensures that they also have proper dimensions.

```
def mult :: forall n : N, m : N, k : N .
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) (Vec (S k) Int) ->
  Vec (S n) (Vec (S k) Int)
def mult a b = map ((flip multVec) b) a
```

```
def multVec :: forall n : N, m : N .
  Vec (S n) Int ->
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) Int
def multVec v m =
  map (foldl1 (\x y -> x + y))
  (map (zipWith (\x y -> x * y) v) (transpose m))

def transpose :: forall n : N, m : N .
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) (Vec (S n) Int)
def transpose matrix =
  let indices = mapi const (head matrix) in
  map (flip column matrix) indices

def column :: forall n : N, m : N .
  Int ->
  Vec (S n) (Vec (S m) Int) ->
  Vec (S n) Int
def column i = map (nth i)

def nth :: forall n : N, a : * . Int -> Vec (S n) a -> a
def nth 0 (x : xs) = x
def nth _ [x] = x
def nth n (x1 : x2 : xs) = nth (n - 1) (x2 : xs)
```

Statically typed printf function

The well-known `printf` function from the C programming language, uses a string to provide formating of a printed text. However this approach has a major drawback, formated arguments are not statically type checked. As a result of that, writing `printf("%d", 3.14);` will print meaningless int, without emiting any warning or error. That's where the GADTs come to the rescue. In the following example, we define `Format` data type which is used to express intended formating of a printed string. By chaining constructors together we define type of intended printing function, which is accumulated in unnamed parameter of the `Format` data type. When a value of the type `Format` is applied to the function `printf` an appropriate printing function is built by step by step deconstruction of the `Format` value. By combining this approach with the function composition operator (.) (for writing more readable chains of constructors), we get a neat and type-safe way of pretty-printing values into the standard output.

```
data Format * where
  | Str :: forall a : * . Format a -> Format (String -> a)
  | Inr :: forall a : * . Format a -> Format (Int -> a)
  | Flt :: forall a : * . Format a -> Format (Float -> a)
  | Bl  :: forall a : * . Format a -> Format (Bool -> a)
  | Chr :: forall a : * . Format a -> Format (Char -> a)
  | Lit :: forall a : * . String -> Format a -> Format a
  | Eol :: forall a : * . Format a -> Format a
  | End :: Format ()

def printf ::  forall a : * . Format a -> a
def printf End = ()
def printf (Lit s format) = putStr s `seq` printf format
def printf (Eol format) = putStrLn "" `seq` printf format
def printf (Str format) =
  \x -> putStr x `seq` printf format
def printf (Inr format) =
  \x -> (putStr . intToString) x `seq` printf format
def printf (Flt format) =
  \x -> (putStr . floatToString) x `seq` printf format
def printf (Bl  format) =
  \x -> (putStr . boolToString) x `seq` printf format
def printf (Chr format) =
  \x -> putChar x `seq` printf format

def main :: ()
def main =
  putStrLn "What is your name ?" `seq`
  let name = getLine () in
  printf ((Lit "Hello " . Str . Lit "!" . Eol .
           Lit "The answer is: " . Inr . Eol) End) name 42
```

Chapter 3

Type system

Chapter 4

Semantics

Chapter 5

Future work