Overview of the Contua programming language.

Jacek Olczyk

Feb 2019

1 Introduction

Contua is a strongly typed functional programming language. It is conceived on a basis of the idea that the language should force the programmer to write all the functions continuation-style. It is not immediately clear what measures needs to be taken to disallow regular functions. I needed to come up with some conventions to achieve that result.

2 The syntax

Since the language will be pretty complicated in the 'backend', I tried to keep the syntax as simple as possible. In this grammar I omitted the whitespace rules.

```
  program = { typeDecl }, { funDecl }
  funDecl = type, '::', id, { id }, "=", expr,";"
 typeDecl = 'type', ID, { id }, "=", typeCtor, { "|", typeCtor },";"
     type = basicType | (type, { '->', type }) | "(", type, ")"
basicType = typeCtor | id | "[", id, "]"
 typeCtor = ID, { type }
     expr =  id | ID | expr, "+", expr | expr, "-", expr | expr, "*", expr | "-", expr |
            | "(", expr, ")" | expr, expr | ('fn'| "\" | "λ"), { id }, ".", expr |
            | 'let', expr , "=", expr , 'in', expr |
            | 'match', expr, 'with', { "|", expr, '=>', expr } |
            | 'if', expr, 'then', expr, 'else', expr | expr, '==', expr |
            | expr, 'and', expr | expr, 'or', expr | 'not', expr |
            | expr, '<=', expr | expr, '>=', expr | expr, '<', expr | expr, '>', expr |
            | "["expr, {",", expr }"]" | expr, ":", expr | expr, '++', expr
```

3 Forcing continuation style

Forcing the continuation style on the programmer is done by adding an implicit last argument and changing the return type of all functions that are defined in the language. For example, if a function is annotated with type `a -> b -> c`, then the actual type of the function is `a -> b -> (c -> d) -> d`, where the result type `d` is always generic - this is the continuation.

4 Language details

The language is mostly a fusion of OCaml and Haskell, with OCaml-style pattern matching and Haskell-style pretty much everything else. The only exception are the boolean operators, which are words (not, and, or, instead of the usual !, &&, ——), and the lambdas, which use a dot (.) to separate the arguments from the body, and 3 different ways to start them: the keyword 'fn' (similar to fun in OCaml), backslash (like in Haskell) or the Unicode lambda.

5 Sample program

```
type Bool = True | False;
type Maybe a = Just a | Nothing;
type Either e a = Left e | Right a;

Int -> Int ::
fac n c =
  if n == 0
    then c 1
    else fac (n - 1) (λf . c (n * f));

Int -> Int ::
fib n c =
  if n <= 1
    then c 1
    else fib (n - 1) (λres . fib (n - 2) (λres2 . c (res + res2)));

Int -> Int -> Int ::
sum a b c = c (a + b);

Int -> Int -> Int ::
max2 a b c = c (if a <= b then b else a);

Int -> Int -> Int -> Int ::
max3 x y z =
  if x <= z
    then max2 y z
    else max2 x y;
```

6 Points

I plan to implement all the features mentioned in the task description required for 35 points except for type synonyms, which are not .