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

```
    program = { typeDecl | funDecl }
    funDecl = [type, ":"], [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 either abstract or decided by an optional explicit annotation with the semicolon - this is the continuation. The programmer must then write the function using said parameter and it is expected that the result of the function is passed to the continuation. The entry point to the program should be defined using a function with the name main which should be of type that can accept an identity continuation, and such continuation will be implicitly passed to it. As such, the only exceptions to the continuation modification are the main function and builtins, which includes both functions defined directly in the interpreter that the operators will be desugared into and the prelude, which will be loaded with the file and written in Contua.

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. The language requires the programmer to specify the type of every top-level function declaration, however full type inference is still performed and the result is compared with the modified type of the function. The language supports recursive polymorphic algebraic types and generic lists, both supporting unbounded pattern depth. The language does not allow for runtime errors, as there is no division operator and the type system will reject any program with non-exhaustive patterns in a match expression. The need for error checking is alleviated by the programmer being able to ignore the passed continuation and instead return a new continuation.

5 Sample programs

```
# prelude.cont
# this file is loaded before all the input files by the interpreter and contains basic
# definitions of standard library functions and types.

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

a -> a ::
id x = x;

Int -> Int ::
succ x = x + 1;

(a -> b) -> (b -> c) -> a -> c ::
comp f g x = g (f x);

(a -> b) -> [a] -> [b] ::
map f xs = match xs with
| [] => []
| h:t => f h : map f t;

(a -> (b -> c) -> c) -> [a] -> ([b] -> c) -> c ::
contMap f xs c = match xs with
| [] => c []
| h:t => f h (\x . contMap f t (\rest . c (x : rest)));
```

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

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

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

```
# Example of throwing an exception.
Maybe Int :
Int -> Int -> Int ::
div x y c =
  if y == 0
    then Nothing
    else safeDiv x y c;

Int -> Int -> Int ::
safeDiv x y c =
  if x < y
    then c 0
    else safeDiv (x - y) y (λrest. c (rest + 1));

(Int -> Int) -> Int ::
main = λc. fib 10 (λfi. fac 10 (λfa. safeDiv fi fa (λd. max3 fi fa (100 * d) c)));
```

```
type Tree a = Empty | Node a (Tree a) (Tree a);

mapTree f t c = match t with
  | Empty => c Empty
  | Node x t1 t2 => mapTree f t1 (λl . mapTree f t2 (λr . Node (f x) l r));

testTree = Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty);

main = mapTree succ testTree id;
```

```
type TwoLists a b = TwoLists [a] [b];

[a] -> Maybe a ::
safeHead xs c = match xs with
| h : _ => c (Just h)
| [] => c Nothing;

(a -> b -> c) -> [a] -> [b] -> Maybe c ::
joinSafeHeads f xs ys c = match TwoLists xs ys with
| TwoLists (h1:_) (h2:_) => comp c Just (f h1 h2)
| TwoLists _ _ => c Nothing;

Maybe Int ::
main = safeHead [2, 1, 3, 7] id;
```

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 necessary and runtime errors which will are not needed in the language, in particular everything already mentioned in the language details. It might be also worth considering to add a do notation like in haskell, but for continuations (it is visible in the main function that it could simplify code greatly), however it was a last-second consideration and if I find the time, I'll do that.