Intérprete de Lambda Cálculo

Deseño de Linguaxes de Programación

Rodríguez Arias, Alejandro
alejandro.rodriguez.arias@udc.es

Bouzas Quiroga, Jacobo
jacobo.bouzas.quiroga@udc.es

Tuesday 12th December, 2017

Contents

1 Introduction

The present work reports the refactoring of the untyped lambda calculus implementation by Benjamin C. Pierce into a toplevel interpreter, executed as course work in the present year.

The rest of the work is structured as follows. The changes made in the original source are described in detail in section 2. Section 3 features an user manual and some execution examples for the untyped lambda calculus interpreter.

2 Changes

The changes introduced in the original source are mostly restricted to the `main.ml` source file. In the first place, the `parseArgs` function and the `argsDef` data structure were modified to provide a command line interface to allow for both interactive execution and the loading of script files. We accomplished this by adding a `-f` option to the interpreter, which takes a file name to be executed as its only argument and respects the interpreter original functionality. The differences in the `parseArgs` and `argsDef` can be seen in figure 1.

After that, a new `toplevel` function was created, with the purpose of showing a promtp, reading user line input, parsing it and executing it. The code for this function is shown in figure 2.

Figure 1: Differences in the `parseArgs` and `argsDef` definitions.

Additionally, an obsolete OCaml construct, the methods `.set` and `.create` on the `String` class were removed and substituted with their corresponding `Byte` class equivalents. This helped avoid some compile-time warnings.

Furthermore, the `process_file` routine was modified to handle execution exceptions by itself, removing the need to do it in the main function, and therefore, freeing it

from that responsibility. This allowed for script and interpreted code to handle these exceptions in different ways: terminating execution on scripts and resuming it on the interpreter.

Finally, the `main` function was modified accordingly to account for this changes. When invoked with no `-f` argument, the toplevel function is called, else, the original execution flow is invoked. Also, various modifications were made in the way the original program handled output to enable the interpreter to display errors and execution results in an appropiate manner.

```
let rec toplevel ctx =
  Printf.printf ":: "; (* prompt *)
  try
    let input = read_line () in
    (* creates a lexer buffer from the input string *)
        let lexbuf = Lexing.from_string input in
        let result = try
      (* parse the lexed input string, return a syntactic tree *)
          Parser.toplevel Lexer.main lexbuf
          (* unless there's an error on the input string *)
          with Parsing.Parse_error as e ->
                  (* pretty print the error and scalate the exception *)
        print_flush();
        open_hvbox 0;
        printInfo (Lexer.info lexbuf);
        print_space ();
        print_string "Parse error";
        print_cut(); close_box(); print_newline();
        raise e;
        in
        (* we apply the semantic tree to a context to obtain a command list *)
        let cmds, _ = result ctx in

        (* this function allows us to evaluate a command in a given context *)
    let g ctx c =
      open_hvbox 0;
      let results = process_command ctx c in
      close_box();
      print_flush();
      results (* return context *)
    in
    (* we evaluate every command on the list on our current
    context while updating it with each execution *)
        let new_ctx = List.fold_left g ctx cmds in
        (* after which, we clear the parser's stack *)
    Parsing.clear_parser ();
    (* and recall the interpreter's main loop with an updated context *)
    toplevel new_ctx
  (* interpreter will exit on EOF (Ctrl+D) *)
  with
    Parsing.Parse_error -> toplevel ctx
    | Support.Error.Exit(code) -> close_box(); print_flush(); toplevel ctx
    | End_of_file -> ()
```

Figure 2: Definition of the `toplevel` function.

3 Manual

Compiling the toplevel

A Makefile is provided to facilitate the task of compiling the interpreter.

```
$ make
```

Executing the toplevel

You can launch the toplevel invoking the executable on the shell

```
$ ./f
```

If the intention is to execute a source file in a non-interactive manner, then the name of such file must be specified following the `-f` option.

```
$ ./f -f an_script.f
```

Using the interpreter

Here are some execution examples in the untyped lambda calculus that the interpreter handles.

Church numerals

```
:: church_zero = lambda f. lambda x. x;
:: church_iszero = lambda n. n (lambda x. false) true;
:: church_succ = lambda n. lambda f. lambda x. f (n f x);
:: church_1 = church_succ church_zero;
:: church_2 = church_succ church_1;
:: church_3 = church_succ church_2;
:: church_4 = church_succ church_3;
:: church_5 = church_succ church_4;
:: church_6 = church_succ church_5;
:: church_7 = church_succ church_6;
:: church_8 = church_succ church_7;
:: church_9 = church_succ church_8;
:: church_10 = church_succ church_9;
:: church_plus = lambda m. lambda n. lambda f. lambda x. m f (n f x);
:: church_mult = lambda m. lambda n. lambda f. m (n f);
:: unchurch = lambda f. f (lambda x. succ x) 0;
```

Interpreter responses are omitted for brevity. Some execution results using these definitions are shown below.

```
:: unchurch (church_pred church_9);
8
:: unchurch church_9;
9
:: unchurch (church_succ church_9);
10
:: unchurch (church_mult church_9 church_7);
63
```

Fixed-point combinator

The fixed point combinator can be implemented in the following way. It must be noted that the user should not call it with only one argument, as it will execute forever in an eternal loop.

```
fix = lambda f. (lambda x. f (x x)) (lambda x. f (x x));
```