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

Wednesday 4th July, 2018

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

We introduced changes in the original source code to provide the programmer with useful functionality like an interactive interpreter, printing of currently defined identifiers, execution traces and recursive functions.

Interactive interpreter

To provide an interactive interpreter we implemented a simple read-eval-print loop using the available facilities of the interpreter. We did this by adding a command line flag to allow for normal functionality of the compiler and then implementing a recursive `toplevel` function that will prompt for user input, read the line introduced by the user, parse it, execute it and print a result on screen.

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 appropriate manner.

List of defined identifiers

Another useful feature for the interpreter is to be able to display a list of defined identifiers that are declared in the current context held by the interpreter. We added a new keyword, `ctx`, in the parser and a new kind of command to the language: debugging commands, that launch special actions when introduced in the interpreter.

To accomplish the listing of defined identifiers we used already existing printing functions and iterated over the context, examining defined bindings and printing them in the same way the compiler/interpreter would print any term. In the case of abstract lambda terms, internal names for variables had to be resolved by term shifting before actually printing them.

Execution trace printing

To closely monitor how the interpreter operates, it is useful to be able to display detailed real-time information about the evaluation of an execution tree. We added two instructions, `start_trace` and `stop_trace`, to the language. These allow setting an internal flag, `__TRACE__`, in the context that when true, enables a printing routine in the evaluation function.

To provide more information about what is going on during an execution, we added a type printing function that can print the type of a given term. In this way, the traces not only show the term itself but its internal type, which can be useful to grasp the meaning of the trace.

Additionally we implemented growing levels of indentation to each level of recursion in the evaluation of a term, thus showing how deep in the evaluation we are at every given moment.

Recursive functions

To allow definition of recursive functions we implemented the fixed-point combinator(FPC) in the language and provided a `rec` keyword that functions similarly to lambda terms but applies the defined function to it.

Implementation of the fixed-point combinator, shown in figure 1, was complicated due to the strict nature of the compiler's evaluation. In strict evaluated languages, an extra dummy parameter is necessary for the definition of the FPC to stop the evaluation from going into an infinite loop when it is declared.

```
fix =  lambda fun.
    (lambda fpc. fun (lambda dummy. fpc fpc dummy))
    (lambda fpc. fun (lambda dummy. fpc fpc dummy));
```

Figure 1: Fixed-point combinator for strict evaluation as implemented in the language

An example of use of the `rec` keyword can be seen in figure 2.

```
/* a simple countdown function that always returns zero */
rec count. lambda n. if iszero n then n else count (pred n);
```

Figure 2: Example of use of the rec keyword

Miscellaneous changes

Additionally to the changes described before, 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.

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 booleans

```
:: /* church booleans, true and false */
:: t = lambda x. lambda y. x;
t = lambda x. lambda y. x;
:: f = lambda x. lambda y. y;
f = lambda x. lambda y. y;
::
:: /* ~a */
:: not = lambda a. f t
not = lambda a. f t
::
:: /* a ^ b (conjunction) */
:: and = lambda a. lambda b. a b a
and = lambda a. lambda b. a b a
::
:: /* a v b (disjunction) */
:: or = lambda a. lambda b. a a b
```

```
or = lambda a. lambda b. a a b
::
:: /* ~a ^ b (implication ->) */
:: lambda a. lambda b. a f b
lambda a. lambda b. a f b
```

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.

```
fix =  lambda fun.
    (lambda fpc. fun (lambda dummy. fpc fpc dummy))
    (lambda fpc. fun (lambda dummy. fpc fpc dummy));
```

Recursive functions Recursive functions can be declared using the fixed point combinator, creating an abstracted function that takes the function that should be called recursively as parameter and letting the fixed-point combinator pass the function itself as that parameter.

This same goal can be accomplished by the use of the rec keyword.

```
/* integer equality */
inteqabs =
        lambda inteq.
        lambda n1.
        lambda n2.
                if iszero n1
                then
                        if iszero n2
                        then true
                        else false
                else
                        if iszero n2
                        then false
                        else
                                inteq
                                        (pred n1)
                                        (pred n2);

inteq = fix inteqabs;

/* or with the rec keyword */
recinteq = rec inteq. lambda n1. lambda n2. if iszero...

/* applied example */
:: reqinteq 31 17;
false
```