`lci` Manual

Kostas Chatzikokolakis

12 March 2006

This manual describes the use of `lci`, which is an advanced interpreter for the λ-calculus. This program was first developed by Kostas Chatzikokolakis as an assignment for the “Theory of Programming Languages” course in the Department of Informatics of University of Athens. Later it became an open source project licenced under GPL, in order to be used and improved by the open source community. It’s main purpose is to compute the normal form of pure λ-calculus terms. Moreover it supports various extensions which are, however, implemented through the pure calculus.

1 Syntax

`lci` supports the syntax of λ-calculus enriched with *integers*, *identifiers* and *operators*. The supported language is described by the following grammar:

$$\begin{array}{rcl} \text{Term} & \rightarrow & \text{var} \\ & | & (\ \text{Term Oper Term}\ ) \\ & | & (\ \lambda\ \text{var}\ .\ \text{Term}\ ) \\ & | & \text{num} \\ & | & \text{id} \\ \text{Oper} & \rightarrow & \text{op} \\ & | & \varepsilon \end{array}$$

The symbol λ can be written as “\” or as the greek character “λ” (using a greek character set). Also “`->`” can be used instead of a dot. **var** and **id** are arbitary strings of latin characters (uppercase or lowercase), numbers and underscores. However **var** must start with a lowercase or underscore, while **id** must start with an uppercase . It is also possible for any character to be used in an **id** if enclosed in single quotes. **num** is a string of numbers and **op** is a string that contains the following characters

```
~ ! @ $ % ^ & * / + - = < > | . , : ;
```

except from the reserved operators `->` `.` `=` `;` `?`. Finally parentheses can be avoided according to the following rules

- Outmost parentheses can be avoided
- Application is left-associative
- The scope of an abstraction extends as far to the right as possible
- Terms that contain operators are parsed according to the precedence and associativity of these operators (see section 6).

2 Basic function

`lci` is an interactive program. When executed it displays the `lci>` prompt and waits for user input. The most simple usage is to enter a λ-term and press return. The program performs all β and η reductions and generates the term's normal form. The result is printed in a "readable" way, that is only the necessary parentheses are displayed, church numerals are displayed as integers and lists using the standard Prolog notation. However the way terms are displayed can be modified, for example the following command

```
Set showpar on
```

causes all parentheses to be displayed.

Terms are reduced using the *normal order evaluation strategy*, that is the leftmost β or η reduction is performed first. This strategy guarantees that term's normal form will be computed in finite time, if it exists. However if a term has no normal form then execution will not terminate. After the execution the program displays the number of reductions that were performed and the CPU usage time.

3 Integers

`lci` supports integers by encoding them as *church numerals* during parsing. Integer n will be converted to

$$c_n \equiv \lambda f.\lambda x.f^n(x)$$

So, actually, it is just a syntactic sugar. All operations are implemented in pure λ-calculus and can be used through identifiers and operators. Although the use of calculus for ordinary operations is sleek, it has has a serious performace drawback. The complexity of an operation is far from constant, for example the power-of operator (`**`) requires an exponential number of reductions. Moreover, due to stack limitaions, the greatest suported integer is 9999.

4 Identifiers

Identifiers are used to represent big λ-terms by defining *aliases*. For example term $\lambda x.x$ can be assigned to alias I so that the term $I\ y$ is equivalent to $(\lambda x.x)\ y$. Aliases must be defined in a file that is read by the program. The syntax of this file is described by the following grammar.

$$\begin{array}{rcl} \text{CmdList} & \rightarrow & \text{Cmd ; CmdList} \\ & | & \varepsilon \\ \text{Cmd} & \rightarrow & \text{id} = \text{Term} \\ & | & \text{? Term} \end{array}$$

Command `id = Term` assings Term to identifier `id` while command `? Term` causes the evaluation of `Term` just as if it was entered interactively. File processing is made using the following command

```
Consult 'file'
```

Morever, when being started, `lci` searches for a file named `.lcirc` in the following places

```
$PREFIX/share/lci/.lcirc  (eg. /usr/local/share/lci/.lcirc)
$HOME/.lcirc
./.lcirc
```

in that order. All files found are executed, if none is found then a warning is printed. This file contains definitions for many basic functions and operators (integer operations, for expample) and is similar to Haskell's `prelude.hs`.

Identifiers are replaced by the corresponding terms during evaluation and *not* during parsing. Thus the order of the definitions is not significant. If an alias is not defined an error message is displayed during evaluation. If no alias contains itself (directly or indirectly) then aliases are just a syntactic sugar, for if we replace all of them we get valid λ-terms. However `lci` supports curcular references of aliases as a way to implement *recursion*. This idea is described in the following section.

5 Recursion

Recursion is an essential programming tool provided by all serious programming languages. `lci` supports two methods of implementing recursion: using *infinite temrs* or using *fixed point combinators.*

5.1 Recursion using infinite terms

If we allow an alias to contain itself then we can write terms like

$$M = \lambda x.M\ y$$

Replacing M according to this definition we get the term $\lambda x.(\lambda x.M\ x)\ x$ and if we keep on replacing we get the term

$$\lambda x.(\lambda x.(\lambda x.(...)\ y)\ y)\ y$$

Thus M can be considered as a term of infinite length. Of course this is not a valid λ-term, however it can be useful.

Now consider an arbitary closed λ-term M, that is $FV(M) = \emptyset$. We can easily infer the following

$$\begin{array}{lcl} M[x := N] & \equiv & M \\ (\lambda y.N)[x := M] & \equiv & \lambda y.N[x := M] \\ \lambda x.M\ x & \rightarrow_\eta & M \end{array}$$

Thus variable substitution as well as η-reduction can be performed without knowing the definition of M, that is without the need to replace it. So during the evaluation of a term, M must be replaced only if we reach one of the following terms:

$$M\ N \tag{1}$$

$$M \tag{2}$$

If the first case M may contain an abstraction and in the second any redex. If we replace an alias only when necessary, we can finish the evaluation without performing all the replacements. For example the abstraction $\lambda x.y$ does not use its argument, so the following reduction

$$(\lambda x.y)\ M \rightarrow_\beta y$$

eliminates M without computing it.

`lci` handles identifiers using this techique. That is it replaces an identifier with it's corresponding term only when necessary and only once at a time. So even if a term is recursive, it is possible to find a normal form if recursion is interrupted by some condition. `.lcirc` contains many recursive definitions, mainly concerning list manipulation functions.

This technique is not compatible with the pure calculus, as it uses invalid λ-terms. However the following must be noted: suppose that in term $M = \lambda x.M\ x$ we need to replace M only twice until we reach it's normal form. This means that in term $\lambda x.(\lambda x.(\lambda x.M\ y)\ y)\ y$ no replacement will be performed. M. So we can substitute M with an arbitary valid λ-term N and we get

$$M' = \lambda x.(\lambda x.(\lambda x.N\ y)\ y)\ y$$

M' behaves exactly like M but it is a valid term. Of course in a different situation more replacements could be needed, producing a different M'. So M could be considered as a "term generator" that produces an appropriate M' each time.

5.2 Recursion using a fixed point combinator

Recursion can be implemented in pure λ-calculus in a very sleek way, using a fixed point combinator. A such combinator Y is a closed λ-term that satisfies the following relation

$$Yf \to f\ (Yf)$$

for any term f. Now let f be a term that satisfies $f = E$, where E is an expression that containts f. We convert f to a variable forming the non-recursive term $F = \lambda f.E$. It is easy to see that the function we seek is a fixed point of F, that is YF.

Things get a little bit more complicated if we have the following set of mutually recursive terms

$$\begin{array}{rcl} f_1 & = & E_1 \\ & \vdots & \\ f_n & = & E_n \end{array}$$

where any f_i can be contained in any E_j. Now, before applying Y, we must join all terms in one. This can be done using the functions TUPLE n and INDEX k. The former packages n terms into a n-tuple, the latter returns the k-th element of a tuple. Both of them can be implemented in pure λ-calculus. So we build the following recursive term

$$f = \text{TUPLE}\ n\ f_i\ \dots\ f_n$$

and replace any occurences of terms f_i using INDEX k

$$f_i \to \text{INDEX}\ i\ f$$

Finally f is defined using a fixed point combinator

$$f = Y\ (\lambda f.\text{TUPLE}\ n\ f_i\ \dots\ f_n)$$

We have already mentioned that `lci` allows an alias to contain itself. The default way of handling such aliases was described in paragraph 5.1. Moreover `lci` provides the command `FixedPoint` which removes circular references from aliases using a fixed point combinator. Initially this command creates a graph in which each vertex corresponds to an alias. Two vertexes S, T are connected with a directed arc if alias S contains T in it's definition. A circle in this graph denote a set of mutually recursive terms. `FixedPoint` detects such circles and, in case they contain more than one arc, it packages the corresponding terms using functions TUPLE and INDEX. Then it removes recursion using the combinator Y which must be defined in `.lcirc`. The modified definition of recursive aliases can be displayed using the `ShowAlias` command.

6 Operators

Operators is another tool that is provided by almost all programming languages. `lci` supports operators as a special kind of function that takes two arguments and syntactically appears between them. Using an operator requires two steps. The first is it's *declaration* together with it's *precedece* and *associativity*, in a way similar to Prolog. This can be done with the command

```
DefOp 'operator' preced assoc
```

Quotes are necessary so that operator's name is recognized as an identifier. Precedence is an integer between 0 and 255 and is used during parsing when no parentheses are present. Associativity takes one of the following values:

`yfx`	Left-associative operator
`xfy`	Right-associative operator
`xfx`	Non-associative operator

Character `x` corresponds to a term with lower precedence than the operator, while `y` to one with higher or equal. Thus expression `a+b+c*d` will be recognized as `(a+b)+(c*d)`, for operator `*` has lower precedence [1] than `+` and `+` is left-associative. Terms that are not the result of an operator, or are enclosed in parentheses, are considered to have precedence 0. Moreover application is considered as a left-associative operator with precedence 100. So if an operator `$` is declared with precedence 110 then the expression `a b$c` will be recognized as `(a b)$c`.

The second step is operator's *definition* which is performed by defining an alias with the same name:

```
'operator' = ...
```

Operator definitions must be placed in a file (as all alias definitions) and quotes are required. During parsing, `lci` replaces operators with identifiers, thas is expression `a+b` will be transformed to `'+' a b`. Now `+` is an identifier, not an operator, and will be replaced with the corresponding term during term's evaluation.

In `.lcirc` many common operators are declared and defined, mainly concerning integers and list manipulation. These include the right-associative operator `:` to write lists as `a:b:c:Nil`, operator `++` to append lists, operator "`,`" to build ordered pairs `(a,b)`, integer operations, integer comparisons etc.

7 Evaluation strategies

An evaluation strategy determines the choice of a redex when there are more than one in a term. `lci` uses the *normal order* strategy, which selects term's

[1] `lci` behaves similarly to Prolog, that is lower precedence operators are applied first.

leftmost redex. The main advantage of this strategy is that it always leads to term's normal form, if it exists. However it has a serious drawback which is the multiple computation of terms. For example in the following series of reductions

$$\begin{array}{ll} & (\lambda f.f(f\ y))((\lambda x.x)(\lambda x.x)) \\ \to & (\lambda x.x)(\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\ \to & (\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\ \to & (\lambda x.x)(\lambda x.x)y \\ \to & (\lambda x.x)y \\ \to & y \end{array}$$

the term $(\lambda x.x)(\lambda x.x)$ was computed twice. An alternative strategy is *call-by-value*, in which all arguments are computed before applied to a function. This method can avoid multiple computation.

$$\begin{array}{ll} & (\lambda f.f(f\ y))((\lambda x.x)(\lambda x.x)) \\ \to & (\lambda f.f(f\ y))(\lambda x.x) \\ \to & (\lambda x.x)((\lambda x.x)y) \\ \to & (\lambda x.x)y \\ \to & y \end{array}$$

This strategy, however, does not guarantee that normal form will be found. There are also some other strategies like *call-by-need* that is used in some functional languages like Haskell.

`lci` does not implement any such technique, but there has been an effort to overcome this problem using a special operator $\sim$. This operator does not behave like ordinary operators. The expression $M \sim N$ denotes the application of M to N which, however, uses call-by-value. So, if $M \sim N$ is the leftmost redex then all reductions of N are performed before the application. Thus the term $(\lambda f.f(f\ y)) \sim ((\lambda x.x)(\lambda x.x))$ will be reduced according to the second of the previous ways. Operator $\sim$ has the same precedence and associativity as the application operator, so it can be easily combined with it.

This operator, however, should be used with caution since the normal form of $(\lambda x.y) \sim ((\lambda x.x\ x)(\lambda x.x\ x))$ will never be found, yet it exists. In file `queens.lci` there is an implementation of the well-known n-queens problem, using experimentally this operator. Without the use of the operator the program is impossible to terminate, even for 3 queens where the combinations that must be examined are very few. This is due to the fact that terms are extremely complex and cause a lot of recomputation. Using the operator $\sim$ and testing in an Athlon 1800, all solutions for the 3 queens where found in 0.3 seconds, for 4 queens in 4.4 and for 5 in 190. For 6 queens after many hours of testing the program did not terminate. This is not strange, though, since Haskell (with the same implementation and using lazy-evaluation and constant time arithmetic) needs 1799705 reductions for the 8 queens and extremely much time for $n > 12$.

8 Tracing

`lci` supports evaluation tracing. This function is enabled using the following command

```
Set trace on
```

or pressing `Ctrl-C` during the evaluation of a term. When tracing is enabled, the current term is displayed after each reduction and the program waits for user input. Available commands are `step`, `continue` and `abort`. The first one performs the next reduction, the second continues the reductions without tracing and the last one stops the evaluation. An alternative function is to display all intermediate terms without interrupting the evaluation. This can be enabled using the following command

```
Set showexec on
```

9 System commands

In previous paragraphs we have already mentioned some commands that are supported by `lci`. These commands are functions that may have arguments. If such a function is the leftmost in a term, then, instead of evaluating the term, a system command is executed. All system commands are described in table 1.

Command	**Function**
`FixedPoint`	Removes circular references from aliases using a fixed point combinator Y
`DefOp op prec ass`	Declares an operator with the given precedence and associativity.
`ShowAlias [name]`	Displays the definition of the given alias, or a lists of all aliases.
`Print term`	Displays a term. Useful to check parsing.
`Consult file`	Reads and processes the given file.
`Set option on/off`	Changes one of the following parameters.
`trace`	Evaluation tracing
`showexec`	Display all intermediate terms
`showpar`	Display all term's parentheses
`greeklambda`	Display "λ" instead of "."
`readable`	Readable display of integers and lists
`Help`	Displays a help message.
`Quit`	Terminates the program.

Table 1: System commands

10 Examples

In this section there are some expamples of using the program.

```
lci> 3+5*2
13

lci> Sum 1..10
55

lci> Take 10 (Nats 5)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

lci> Map (Add 3) 1..5
[4, 5, 6, 7, 8]

lci> Map (\n.n**2) 1..5
[\y.y, 4, 9, 16, 25]

lci> Filter (Leq 6) 3:6:10:11:Nil
[6, 10, 11]

lci> Length 1..10 ++ 4:5:Nil
12

lci> (Member 3 1..10) && (Length 3:4:5:Nil) >= 3
\x.\y.x
```

Note that term $\lambda y.y$ is the normal form of number 1. In file `queens.lci` there is an implementation of n-queens problem.

```
lci> Consult 'queens.lci'
Successfully consulted queens.lci
lci> Queens 4
[[2, 4, \y.y, 3], [3, \y.y, 4, 2]]
```

All of the above functions can be also evaluated using the `FixedPoint` command, which removes circular references using the fixed point combinator Y. Using `ShowAlias` you can see an alias definition after the modification.

```
> FixedPoint
> ShowAlias Sum
Sum = Y \_me.\l.If (IsNil l) 0 (+ (Head l) (_me (Tail l)))
```