

The Jivaro head reduction machine for the λ_c -calculus

Version 0.9

Alexandre Miquel

February 2009

Contents

1	Introduction	2
1.1	License	2
1.2	Availability	2
1.3	Credits	2
2	Usage	2
3	Syntax	3
3.1	Lexical conventions	3
3.2	Terms of the λ_c -calculus	3
4	Commands	4
4.1	Define <i>ident</i> = <i>term</i> ;;	4
4.2	Environ ;;	4
4.3	Eval <i>term</i> ;;	4
4.4	Extern <i>ident</i> = <i>string</i> ;;	4
4.5	Help <i>topic</i> ;;	5
4.6	Include <i>string</i> ;;	5
4.7	Load <i>string</i> ;;	5
4.8	Print <i>ident</i> ;;	5
4.9	Quit ;;	5
4.10	Reset ;;	5
4.11	Save <i>string</i> ;;	5
5	Evaluation	5
6	Primitives	6
6.1	Control primitives	6
6.2	Arithmetic operations	6
6.3	String operations	7
6.4	Miscellaneous	7

1 Introduction

Jivaro is a small evaluator for the λ_c -calculus, an extension of λ -calculus with control operators. Inside the Jivaro toplevel, you can build and evaluate programs written in the λ_c -calculus, while defining new instructions and testing them interactively. Jivaro can also be used in batch mode to compile source files and build libraries, or to inspect the contents of compiled files.

1.1 License

Jivaro is copyright © 2009 Alexandre Miquel.

Jivaro is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

1.2 Availability

Jivaro is distributed on the author's web page at

<http://perso.ens-lyon.fr/alexandre.miquel/jivaro/>

1.3 Credits

The Jivaro head reduction machine has been designed and written by Alexandre Miquel (Alexandre.Miquel@{ens-lyon.fr|pps.jussieu.fr}).

The λ_c -calculus has been introduced by Jean-Louis Krivine to extend the theory of realizability to classical logic. The idea of enriching λ -calculus with control primitives comes from Matthias Felleisen, and its connection to classical logic is due to Timothy Griffin. Finally, the λ -calculus itself is the great invention of Alonzo Church.

2 Usage

Usage: `jivaro [options ...] file ...`

where options are:

- batch** enter batch mode (implies `-no-ledit`)
- compile file** compile source file *file* (implies `-batch`). This option does not affect the current environment
- dump file** inspect the contents object file *file* (implies `-batch`). This option does not affect the current environment
- include file** textual inclusion of source file *file* in the toplevel
- init file** specify the location of the initialization file. Default location of this file is `$(JIVARO)/init.lco`
- load file** load the contents of object file *file* into the current environment
- no-init** do not load initialization file

-no-leadit do not start ledit

-help, --help display this help

Anonymous arguments that appear in the command line are treated as file arguments of `-include` or `-load` options depending on their file extension: `.lc` (*lambda-calculus*) for source files, `.lco` (*lambda-calculus object*) for compiled files.

At startup time, an initialization file of primitive definitions is automatically loaded unless the option `-no-init` is given. The default location of this file (that can be overridden with option `-init`) is `$JIVARO/init.lco`, where `$JIVARO` is the installation directory of Jivaro. When Jivaro is run in batch mode only using `-compile` and/or `-dump` options, no initialization file is loaded.

3 Syntax

3.1 Lexical conventions

A comment is introduced by the two-character sequence `(*` and terminated by the two-character sequence `*)`. Comments can be nested, but are never parsed inside string literals. A comment is lexically treated as a blank.

An identifier (*ident*) is any non-empty sequence of characters that contains no space, no delimiter (parentheses, square brackets and curly braces), no backslash, no semi-colon and no double quote. As a special case, a non-empty sequence of digits possibly preceded by a plus (+) or minus (-) sign is not considered as an identifier, but as an integer literal.

An integer literal (*int*) is any non-empty sequence of digits possibly preceded by a plus (+) or minus (-) sign.

A string literal (*string*) is any sequence of printable characters surrounded by double quotes (`"`). As usual, double quote (`"`) and backslash (`\`) need to be escaped using a backslash within a string literal, thus being written `\"` and `\\`. Other characters can be accessed via the escape sequences `\n` (newline), `\r` (carriage return), `\t` (tabulation), `\b` (bell), etc.

Reserved tokens are `;` (semicolon), `;;` (double semicolon), `=` (equal) and `\` (backslash, for the λ). The character sequences `Define`, `Environ`, `Eval`, `Extern`, `Help`, `Include`, `Load`, `On`, `Off`, `Print`, `Quit`, `Reset` and `Save` are treated as keywords when involved in the formation of a command, but one can still use them as regular identifiers in terms.

3.2 Terms of the λ_c -calculus

Terms of the λ_c -calculus are typed using the following BNF:

<i>term</i>	<code>::=</code>	<i>ident</i>	(variable or instruction)
		<i>int</i>	(integer literal)
		<i>string</i>	(string literal)
		<code>\ident term</code>	(λ -abstraction)
		<i>term term</i>	(application)

Application is left-associative and has higher precedence than abstraction.

4 Commands

Each command starts with a capitalized keyword followed by zero, one or several arguments and is terminated by a double semicolon (;).

4.1 Define *ident* = *term* ;;

defines a new instruction *ident* as an alias for the term *term*.

Note that defined instructions are only unfolded when they appear in head position, never in argument position. This design choice may cause some misunderstandings. For instance the sequence of commands

```
Define foo = 100 ;;
Eval int_succ ; foo ; print ; stop ;;
```

is incorrect, since the primitive `int_succ` expects an integer literal on the top of the stack, not an instruction. Rule of the thumb: never use `Define` to define aliases for *int* or *string* literals. Instead, do:

```
Define foo = int 100 ;; (* same as \x x 100 *)
Eval foo ; int_succ ; print ; stop ;;
```

(On the other hand, string and integer literals are handled correctly when they are bound to an *ident* using a λ -abstraction.)

Variant: Define *ident ident*₁ \dots *ident*_{*n*} = *term* ;;

defines a new instruction *ident* that takes *n* arguments on the top of the stack and binds them to the identifiers *ident*₁ \dots *ident*_{*n*} before evaluating the term *term*. This command merely acts as the command

```
Define ident = \ ident1  $\dots$  \ identn term ;;
```

except that the defined constant *ident* is not unfolded when it is evaluated in a stack containing less than *n* elements.

4.2 Environ ;;

lists all the symbols that are declared in the current environment.

4.3 Eval *term* ;;

evaluates term *term* in an empty stack.

Variant: Eval *term* ; *term*₁ ; \dots ; *term*_{*n*} ;;

evaluates term *term* in a stack containing *n* terms *term*₁ \dots *term*_{*n*}.

4.4 Extern *ident* = *string* ;;

defines a new instruction *ident* as an alias for the internal primitive *string*.

4.5 Help *topic* ;;

displays a help message on *topic*, where *topic* is one of `Syntax`, `Commands` or one of the commands `Define`, `Environ`, `Eval`, `Extern`, `Help`, `Include`, `Load`, `Print`, `Quit`, `Reset` or `Save`.

4.6 Include *string* ;;

performs a textual inclusion of file *string* in the evaluator.

4.7 Load *string* ;;

loads a set of declarations from a binary object file *string*. Each declaration of a symbol in the file silently overwrites the (possibly) existing declaration of this symbol in the current environment.

4.8 Print *ident* ;;

prints the declaration of instruction *ident*.

4.9 Quit ;;

exits the evaluator. This command has the same effect as `Control-D`.

4.10 Reset ;;

resets the environment to its initial state. All the declarations of symbols that were not already present in the initial state are lost.

4.11 Save *string* ;;

saves the current environment in the object file *string*.

5 Evaluation

Evaluation implements *head reduction*, a.k.a. the standard (or normal) strategy of the λ -calculus. At each evaluation step, the evaluated process consists of a term called the *head* in front of a stack formed by zero, one or several terms, called the *arguments*. Evaluation proceeds depending on the shape of the head:

- When the head is *\ident term* (abstraction), the topmost argument is popped from the current stack and bound to *ident*. Evaluation then proceeds with head *term* in the new environment. In front of an empty stack, evaluating an abstraction raises exception `stack underflow`.
- When the head is *term₁ term₂* (application), *term₂* is pushed on the top of the stack and evaluation proceeds with head *term₁*.

Primitives that implement arithmetic comparisons take one or two *int* argument(s) on the top of the stack, plus two objects *obj₁* and *obj₂* that respectively represent the ‘then’ and ‘else’ branches of the conditional. When the test succeeds (resp. fails), *obj₁* (resp. *obj₂*) comes into head position.

<code>int_null</code> * <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	nullity
<code>int_eq</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	equality
<code>int_ne</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	inequality
<code>int_le</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	less than or equal
<code>int_lt</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	less than
<code>int_ge</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	greater than or equal
<code>int_gt</code> * <i>int</i> ; <i>int</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	greater than
<code>int_print</code> * <i>int</i> ; <i>obj</i>	<i>obj</i> *	print <i>int</i>
<code>int_read</code> * <i>obj</i>	<i>obj</i> * <i>int</i>	read <i>int</i>
<code>int_userfun</code> * <i>string</i> ; <i>int</i> ; <i>obj</i>	<i>obj</i> * <i>int</i>	pass to <i>obj</i> the image of <i>int</i> argument by the user function of name <i>string</i> . User is prompted to give values of the function at first request

6.3 String operations

<code>string_make</code> * <i>int</i> ; <i>int</i> ; <i>obj</i>	<i>obj</i> * <i>string</i>	string creation
<code>string_length</code> * <i>string</i> ; <i>obj</i>	<i>obj</i> * <i>int</i>	string length
<code>string_get</code> * <i>string</i> ; <i>int</i> ; <i>obj</i>	<i>obj</i> * <i>int</i>	successor
<code>string_sub</code> * <i>string</i> ; <i>int</i> ; <i>int</i> ; <i>obj</i>	<i>obj</i> * <i>string</i>	substring
<code>string_concat</code> * <i>string</i> ; <i>string</i> ; <i>obj</i>	<i>obj</i> * <i>string</i>	concatenation
<code>string_eq</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	equality
<code>string_ne</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	inequality
<code>string_le</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	less than or equal
<code>string_lt</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	less than
<code>string_ge</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	greater than or equal
<code>string_gt</code> * <i>string</i> ; <i>string</i> ; <i>obj₁</i> ; <i>obj₂</i>	<i>obj_{1 2}</i> *	greater than
<code>string_print</code> * <i>string</i> ; <i>obj</i>	<i>obj</i> *	print <i>string</i>
<code>string_read</code> * <i>obj</i>	<i>obj</i> * <i>string</i>	read <i>string</i>

6.4 Miscellaneous

<code>print</code> * <i>obj₁</i> ; <i>obj₂</i>	<i>obj₂</i> *	print <i>obj₁</i> on standard output
<code>flush</code> * <i>obj</i>	<i>obj</i> *	flush standard output