PROSPECTRA
an ESPRIT Project

User Manual for the TrafoLa-H-ML System

Version 1.5

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Deliverable Item S.1.6-DI-15.0

1988-09-05

Distribution: public

ABSTRACT

The backtracking version of TrafoLa-H is a relational language where each expression may have several or even infinitely many values, and each ‘function’ may produce many results. An interpreter for this language was written in Standard ML with a front end in C. This paper describes the usage of the interpreter.

Public

Reinhold Heckmann
Universität des Saarlandes

in the Project

PROgram development by
SPECification and
TRAnsformation

sponsored by the

Commission of the European Communities

under the

European
Strategic
Programme for
Research and development in
Information
Technology

Project Ref. No. 390

Table of contents:

Introduction

This manual refers to the ML interpreter of the backtracking i.e. relational dialect of TrafoLa-H. It must not be confused with the interpreter written in SSL for the functional version of TrafoLa-H.

The described interpreter has version number 1.5. Version 1.5 differs in many details from its predecessor, version 1.4b, that was distributed in June 1988 at Madrid. The system need no longer be loaded into the working directory. The overall execution is much faster (by a factor of 20 for some examples). Some bugs and deficiencies of the older version are now removed, but there is still need for further improvements.

1. General system structure

The TrafoLa interpreter consists of two main parts:

1) Front end (C) = scanner and parser for concrete TrafoLa syntax, and filter for output of the real interpreter

2) Interpreter (ML)

Both programs are running as independent UNIX processes. They are connected via two pipes. The user only interacts with the front end, and all input / output of the real interpreter is filtered through it.

The TrafoLa-H system resides in four directories:

- c.src containing the sources and object files of the front end
- ml.src containing the sources and the object file of the real interpreter
- system containing the files needed to run TrafoLa
- examples containing files with TrafoLa-H programs

The directory 'system' contains:

- trafola the code of the front end that automatically invokes the ML machine
- fam 'functional abstract machine'
  this is the ML machine interpreting abstract ML code
- trafola.ml abstract ML code for the TrafoLa interpreter to be executed by program 'fam'. It is the result of compiling the ML sources specifying the interpretation of TrafoLa
- dequote an auxiliary program needed when PAnndA terms are exported. It should be integrated into 'trafola'
- exrc an editor start up file to be read by 'vi' when it is invoked

The user only needs to know 'trafola', the other ones are automatically executed resp. read when they are needed.

2. Installing the system

The interpreter must know where its system files are to be found. Thus, the environment variable '**TRHML**' must be defined to be the full path name of the system directory. In the PROSPECTRA system distributed by Alcatel in July 1988, this is the directory

$PROSPECTRA/src/trafolah.ml/system

At a SUN work station, the interpreter is immediately executable once the environment variable 'TRHML' was defined by 'setenv'. It is invoked by '$TRHML/trafola', or by simply 'trafola' if a link or alias was provided or '$TRHML' was included in the search path.

In the working directory i.e. the directory where the interpreter is invoked, it creates files whose names consist of the prefix 'tr' followed by a big number derived from host id and process id, and a file with the fixed name 'tmp'. Thus, you should not invoke two instances of the interpreter in the same directory, and make sure that you don't use such a file - it will be overwritten.

In version 1.4, all system files had to be loaded into the working directory, and the files 'tr...' were created in the directory '/tmp'. This was changed since a 'mv' from '/tmp' to the working directory - this was sometimes done by the interpreter - involves a copy and takes much more time than a 'mv' within the working directory.

3. Calling the interpreter

The interpreter is called by

trafola [-i] [file]

The option '-i' suppresses the call of the ML machine. The front end then operates alone and reacts normally, but TrafoLa input is not evaluated. This option is of little use for users.

The file argument gives a 'log and replay' facility. If the named file exists, it is first read and evaluated, then further user input is stored in it. If it does not exist, it is created and all user input is saved in it. If no file is given as argument, the user input is saved nevertheless on a file with prefix 'tr'. Upon leaving the interpreter, the user can decide to remove this file or to rename it into a new TrafoLa file.

4. Leaving the interpreter

The interpreter is left by hitting the EOF character (mostly '^D') when it expects a user input in the main loop, i.e. when the prompt '- ' appears.

If it was invoked with a file argument, the user input during the run is saved there, and the interpreter exits immediately. If the interpreter was not invoked with a file argument, it says

Please give file name to save:

If you only hit 'RETURN', it will exit immediately without saving. If you want your input to be saved, you should enter a file name followed by 'RETURN'. The interpreter then checks whether this file exists and is not a directory. If it is a directory, it asks for a new file name. If it exists, it asks

File exists - overwrite?

Then hit 'y' or 'n'. In the second case, it asks for a new file name. Note that 'y' resp. 'n' should not be followed by 'RETURN', they are accepted immediately when they are typed.

5. The interpreter main loop

The interpreter prompts for input by '- '. When the user gives some input, but not a complete command, additional input is prompted for by '= '. When a complete command is read, it is executed and the result is displayed on screen. Sometimes, the garbage collector of the ML machine is running. It announces itself by the message '[GC: copying ...', and finally gives some information about used space.

There are two kinds of commands: interpreter commands and TrafoLa input. The user is not allowed to give more than one command per line. TrafoLa input may be spread over several lines.

6. Interpreter commands

Interpreter commands are introduced by '#' and consist of a command character and up to two arguments. The arguments are variable or file names. Also absolute and relative path names are legal as file names; relative path names refer to the working directory.

If you don't remember the number or nature of arguments, type only the command name followed by 'RETURN'. The interpreter then asks you explicitly for the arguments. Note that interpreter commands don't end with ';' as TrafoLa input does.

#r <file> read file

The input is switched to the given file. This file should contain a sequence of commands written in exactly the same manner as in ordinary input. It may contain again a '#r' command. The commands given in the file are echoed on the screen preceded by a prompt '< ' (this is a difference to ordinary input) and executed as soon as they are read. When all commands are executed, input switches back to the previous input file.
When a command execution involves user interaction to regard its results, the interpreter expects the required user input always from the keyboard, never from the file just being read.
There is no means to interrupt reading even if the file contains no TrafoLa at all. The interpreter does not accept commands from the keyboard before the file is completely read. Non-terminating

computations resulting from the interpretation of a command in the file may be canceled by '^C' (see below). The interpreter then proceeds by reading the next command from the file.

EOF resp. ^D — end of file

When 'EOF' is encountered during reading from a file, input switches back to the previous input file. When it is encountered during reading from the primary input file i.e. 'stdin', the interpreter terminates (see section 4).

#i <variable> <file> — import object

The given file must contain an object in Cornell structured form, e.g. a PAnndA program. It is translated into a TrafoLa object and bound to the given variable. Thus, it is conceptually equivalent to

dec <variable> = quote (content (<file>))

Note that the import is rather slow. It cannot be interrupted; worse, a trial to interrupt might be dangerous here and mess up the interpreter.

#e <variable> <file> — export object

The value bound to the variable is translated into Cornell structured form and written onto the given file. The export is rather quick and terminated with a message 'Value exported' or 'Export failed'. But note that there are cases where the export fails, but the interpreter tells 'Value exported' after having issued error messages. Exports cannot be interrupted; worse, interrupts might be dangerous here.

#l — list user defined variables

All user defined variables are listed together with the values they are bound to - as far as those values can be displayed. There is no possibility to list the predefined variables.

#k <variable> — kill variable binding

When a user defined variable resp. its value is no longer needed, it may be killed by this command. Only user defined bindings may be killed. If a user defined binding is killed that covered a predefined binding, the predefined one reappears.

Examples:

dec a = 1;		
a;	→	1
dec a = 2;		
a;	→	2
#k a	→	Binding killed
a;	→	Error: unbound identifier 'a'
		(the old binding of 'a' to '1' does not reappear)
Hd [1, 2, 3];	→	1 (predefined)
dec Hd = Tl;		
Hd [1, 2, 3];	→	[2, 3]
#k Hd	→	Binding killed. Predefined meaning reappears
Hd [1, 2, 3];	→	1

#c — collect garbage

The garbage collector of the ML machine is invoked. This command is not really needed, since the garbage collector is invoked automatically during normal execution, but it gives informations about usage of space.

#v — 'vi' call

The screen oriented editor 'vi' is invoked with the actual contents of the log file i.e. with the input typed in so far. Upon start up, the editor learns three new commands (see below). The cursor is placed at the last line of input, and this line is positioned at the bottom of the screen, provided that there is more than one screen of input so far. Besides the usual editor commands, there are three new ones available:

v move back to the beginning of a TrafoLa declaration

V move forward to the beginning of a TrafoLa declaration

K leave ‘vi’ and return to the interpreter. The line where the cursor stood when the ‘K’ command was given and all following lines are again read and evaluated by the interpreter. This implies that you should issue the ‘K’ command only at the first line of an interpreter input or at a blank line between two inputs or after the last input (then nothing is re-read).

It is tolerated, but not recommended to leave ‘vi’ with a command other than ‘K’. The interpreter then issues the message

No re-evaluation of previous input

If the editor is left by ‘:q!’, all changes to the log file are lost. If it is left by ‘:wq’ or ‘:x’, the changes are preserved in the log file, but are not known to the interpreter.

The user is him/herself responsible for giving the ‘K’ command at such a line that all updates are re-read by the interpreter.

Example:

dec fac n = if n == 0 then 1 else n * fac (n-1);
fac 4; → Error: unbound identifier ‘fac’

This is a common mistake; the factorial function is recursive i.e. the declaration should have been introduced by ‘rec’ instead of ‘dec’. Due to this mistake, the occurrence of ‘fac’ at the right hand side is unbound (this is not checked at declaration time).

Correction: Enter ‘#vRETURN’, wait until the editor has been started, hit ‘v’ to move back to the declaration, then hit ‘rr’ to replace the letter ‘d’ by ‘r’, and finally hit ‘K’.

The interpreter then reads the corrected declaration of ‘fac’ and the call ‘fac 4’ again and answers ‘24’.

The keys ‘v’, ‘V’, and ‘K’ were used since they have normally no meaning in ‘vi’. But if you gave them a meaning by ‘EXINIT’ or ‘.exrc’, your meaning is overwritten by the interpreter’s. To cope with this, you may change your ‘vi’ maps or the interpreter’s maps that are defined in the file ‘$TRHML/exrc’.

ˆC (the Unix ‘intr’ character)

This is not an ordinary command but may be given when the interpreter performs a long or never ending computation. This computation is interrupted, and the next command may be entered.

The interpreter acknowledges the interrupt by printing ‘Interrupt’ and displaying a new prompt ‘- ’.

Remarks:

- Whereas on Vax interrupts are accepted immediately, often two or more signals are needed on Sun.
- But don’t be too quick in repeating the signal; otherwise the ML machine may break down with the message ‘Interrupt handler canceled’. The front end often needs some further input until it notices that the ML machine is dead. Then it writes ‘Broken pipe to ML’ and starts the normal exit procedure (see section 4).

Thus, you should hit ‘ˆC’, wait a little bit (a few seconds), then hit it again if the interpreter didn’t stop meanwhile etc.

7. Input, output, and handling of TrafoLa-H values

In relational TrafoLa-H, several classes of values occur as values of expressions and results of functions:

- basic values (Booleans, numbers, constructors, holes, strings)

- structural values (sequences and trees)
- abstract values (functions, macros, and streams)
- pseudo values (not yet evaluated expressions)
- error values (exceptional values carrying a string, the error message)

Expressions evaluate to a multi-set of values that may be empty, contain one element, or a finite or infinite number of elements. Such multi-sets of values are also called multi-values. They are displayed interactively in some order (see following sections).

We shall now enumerate the various classes of values and describe how they may be typed in by the user and how they are displayed by the interpreter.

7.1. Basic values

Basic values are stored and handled as atomic entities although there are means to partition them into subunits e.g. the characters of a string. They are directly mapped onto corresponding ML types (holes to integers).

Booleans: There are only two Booleans that may be typed in as ‘false’ or ‘FALSE’, and ‘true’ or ‘TRUE’. They are displayed as ‘false’ resp. ‘true’.

Numbers are integers 0, 1, 2, etc., and -1, -2, etc. They are displayed as ‘0’, ‘1’, ‘-1’, etc., but only the positive numbers may be directly entered as ‘0’, ‘1’, etc. Negative numbers may be obtained by using the unary ‘-’ operator.
The size of numbers is bounded by the underlying ML system and is likely to depend directly on the hardware. On SUN work stations, the maximal number seems to be $2^{30} = 1\ 073\ 741\ 824$.

Holes are objects @1 @2 @3 etc., i.e. there is a one-to-one correspondence between holes and integers greater than 0. The first hole @1 may be entered as ‘@1’ and also as ‘@’ for abbreviation. It is always displayed as ‘@’.

Constructors are special identifiers declared with ‘con’. They are entered and displayed by their name.

Strings are entered and displayed as (almost) arbitrary sequence of characters enclosed by double quotes ‘"’:

"" "PROSPECTRA" "This is a string."

A back-slash ‘\’ must be entered as doubled back-slash ‘\\’. It is displayed as single back-slash e.g.

Input: "abc\\\\xyz" → *Output:* "abc\\xyz"
Input: Len "a\\\\b" → *Output:* 3

There is no possibility to include a double quote into a string.

Don’t use control characters, single back-slashes ‘\’, or new-lines inside strings. In best cases, you will see an error message, in worse cases, there will be strange effects, and in the worst case, an unrecoverable deadlock results (use control-Z and ‘kill’, or control-\).

Reasons for these effects are a bug in the TrafoLa-H scanner, the behavior of the ML scanner, the special meaning of some control characters for the synchronization of the communication between front end and ML machine, and the fact that displayed control characters may mess up the terminal.

7.2. Structural values

Structural values are the main subject of pattern matching. They may be decomposed by patterns into their constituents, and created by corresponding expressions. They are printed exactly when they are the result of expressions. This is not as good as it sounds, since even huge trees are printed completely.

Sequences consist of an ordered finite collection of other values and are printed as $[v_1, ..., v_n]$. They may be entered using the sequential expression $[e_1, ..., e_n]$.

Examples: [] [1] [[]] [1, "a", true] [1,2,3,4,5,6,7]

Trees consist of a constructor denoting the root label of the tree, and an arbitrary value denoting the children list. This value will mostly be a sequence. Trees may be entered by applying the constructor to the children value.

Examples: con add, op, a, b, c;
add [1, 2]   op true   a (b c)

7.3. Abstract values

Abstract values can neither be entered directly, nor be displayed completely. They may be obtained as result of expressions, e.g. functions as result of pattern abstractions or functional forms.

Functions and **macros** are objects that may be applied to a value resp. a multi-value and then produce a multi-value (see 3.3). All functions are displayed as ‘function’, and all macros as ‘macro’. We agree that this is not too informative. User defined functions and macros may be regarded by looking at the input typed in so far. This is possible by entering ‘vi’ with ‘#v’, searching the last declaration of the function by ‘?<function name>’, and finally leaving ‘vi’ by the command ‘:q’.

Streams are created by freezing the evaluation of an expression (<<e>>), and thus catching a multi-value (the result of this evaluation) into one value. They may be considered as finite or infinite lazy sequences.

Streams are displayed according to the knowledge of the system about their contents. Generally, they are only partially evaluated, and the display routine does not enforce their evaluation. Typical outcomes are

<<...>>	The system knows nothing about the stream.
<<1 $ 2 $ ...>>	The first two items are known to be ‘1’ resp. ‘2’. The remainder is unknown.
<<1 $ 2 $ 3>>	The stream is finite; its three items are ‘1’, ‘2’, and ‘3’.
<<>>	The stream is empty.

The pair ‘<<’, ‘>>’ is used as delimiter since it is an operator denoting the transformation of a multi-value into a stream. ‘$’ is used as separator because it is the operator of non-deterministic choice. One way to produce a finite stream out of ‘1’, ‘2’, and ‘3’ is indeed by writing ‘<<1 $ 2 $ 3>>’.

7.4. Pseudo values

Pseudo values result from recursive or lazy declarations or abstractions and are not yet evaluated. The system does not know anything about pseudo values, not even their type since there is no static type checking. They are displayed as ‘not yet evaluated’.

Examples:

```
- rec fac n  =  if n == 0 then 1 else n * fac (n-1);
> fac : not yet evaluated

- fac;                     (the pseudo value bound to ‘fac’ is evaluated to a
> function                  function after it was read from the environment)

- fac 3;
> 6

- rec a = 1;
> a : not yet evaluated

- a;                       (here, the recursive object evaluates to a number)
> 1
```

7.5. Error values

Error values are not ordinary values. They result from erroneous evaluations and mostly correspond to run-time type errors. Each error value carries a string, the error message. Error values are displayed as ‘Error: s’ where ‘s’ is the message.

Examples:

1 / 0	→	Error: 1 / 0
1 2	→	Error: num applied

if 1 then 1/0 else 1 2	$\rightarrow$	Error: if num then ...

dec pred x = if x == 0 then Error "pred 0" else x-1;

pred 2	$\rightarrow$	1
pred 0	$\rightarrow$	Error: pred 0
pred "a"	$\rightarrow$	Error: string - num

In rare cases, a second kind of error values appears that is displayed as 'Failure: ...'. They result from run-time errors of the ML program implementing the interpreter. The most common reason is the overflow of arithmetic operations. Another, less frequent reason is the handling of interrupts. Other reasons for such errors should not occur if the interpreter is correct, and I think it is.

8. Viewing results

The result of the evaluation of a declaration is a multi-set of environments, whereas the result of evaluating an expression is a multi-set of values. These multi-sets are computed incrementally by backtracking; they may be infinite.

If the result is empty, the interpreter displays '> No' and waits for the next input. If the interpreter surely knows that the result has exactly one element, it displays this element preceded by '> ' and waits for further input.

Otherwise, it displays the first result and silently asks the user whether he/she wants to see the next result. This question cannot be seen on the screen, it is indicated by the fact that the cursor remains just after the displayed thing without skipping to the beginning of the next line and that the prompt '- ' doesn't appear.

The user may then hit some keys to answer the question of the interpreter. The key strokes are noticed immediately, they need not be entered by 'RETURN'.

Possible key strokes after a result has been displayed:

'q'	(quit)	stop showing results
'RETURN'		same as 'q' if you like Prolog
any other key		display the next result. Personally, I like ' ', but Prolog fans will like ';'

After having displayed the last result, there are two possibilities:

First, the interpreter knows without additional computation that this was the last one. Then, it will immediately proceed to display 'No' and to wait for the next command without any user interaction.

Or, the interpreter does not know that it was the last result. Then it silently asks the user whether to produce the next result as described above. If the user confirms, the interpreter will see that there is no more result and display 'No'.

If the results are environments, the global environment is augmented by the bindings in the first environment if there is one; otherwise it is unchanged. If the results are values, they are together bound to the variable 'It' such that they may be considered again or processed by further commands.

Examples:

Input:	1 $ 2 $ 3;		
Output:	1	*Input:* ;	*(next result, please)*
Output:	2	*Input:* q	*(no more results, please)*
Output:	- *(prompt indicating waiting for next TrafoLa input or command)*		

Input:	dec (a . b) = "ML";		
Output:	a : ""		*(first result)*
	b : "ML"	*Input:* ;	
Output:	a : "M"		
	b : "L"	*Input:* ;	
Output:	a : "ML"		*(this is the last result ...)*
	b : ""		

No *(... and the interpreter knows it)*

New bindings induced by first solution:
a : "" *(repeating the first result that binds 'a' and 'b')*
b : "ML"
- *Input:* b; *('b' was bound by the first result)*
Output: > "ML"
-

In version 1.4, there were more possibilities to process the results of a TrafoLa command. They were dropped since the user input was not saved in the log file, but could influence the binding performed by the command. In addition, they required that the user was always consulted after the output of the last result, and even after the output of the final 'No'.

9. Known bugs and deficiencies

The following enumeration does not pretend to be complete. All users are invited to find further bugs or to complain about further deficiencies.

9.1. The overall system

- The system uses and overwrites a fixed file 'tmp' in the working directory.
- There should be more options to control the general behavior.
- There is no means to save compiled TrafoLa functions (as ML abstract code); only the textual form may be saved.
- There are differences between the language of this interpreter and the language of the interpreter written in SSL. Naturally, the main difference is the absence of backtracking there.
- There is no means for direct communication with other PROSPECTRA tools.
- There is no interface to the verifier.

9.2. Interpreter commands

- The '#l' command only shows user defined variables. There is no means to see the predefined bindings optionally.
- Reading a file with '#r' cannot be interrupted or aborted even if the file contains syntax errors, even if the file contains no TrafoLa at all.
- Importing a value with '#i' also cannot be interrupted. Furthermore, it is too inefficient.
- Exporting a value with '#e' is not well integrated from an implementation point of view. But at least, it executes fast.

The editor interface ('#v'):

- There are people who don't like or don't know the editor 'vi'. There should be means to integrate another editor instead.
- The commands 'V' and 'v' may be fooled in some situations.

9.3. The operational behavior

- Often, the interpreter knows that there is only one result, and simply displays it. But sometimes, there is only one result, and the interpreter does not know this.
- A looping subexpression prevents other subexpressions from producing results.
- Interrupts given by '^C' are not reliable.

Suspending and resuming:

The interpreter may be suspended by '^Z' (or your personal suspend character) at any time. But:

- After resumption by ‘fg’ or a similar command, the user cannot see which is the actual state, whether input is expected, and what kind of input. But this is common for Unix programs except ‘vi’.
- When the user regards the results of an evaluation or has to answer some special questions, the input is changed to ‘cbreak’ mode i.e. each key stroke is immediately accepted. If the interpreter is suspended in this state, the terminal mode is messed up. With ‘csh’ and variable ‘filec’ set, the shell behaves normally (this is good), but after resumption, the mode has changed to ‘cooked’ (this is bad). Without ‘filec’, the suspension does not affect the mode i.e. special characters such as the erase character don’t work in the shell (this is bad). But resumption also does not affect the mode such that the TrafoLa-H system continues with the correct mode.

9.4. Input and output

- The secondary prompt only appears when the first important token was given. It is missing if only blank lines and comments are entered.
- There cannot be two inputs on the same line; the second is ignored then (at least, the user is informed about this).
- There is a maximum line length for input lines (255).
- There are some restrictions for strings that should either be overcome, or the user should be hindered to enter such strings.
- Structural values are always completely displayed. The user should be able to specify a maximum depth and/or breadth for their display.
- The external representation of abstract and pseudo values is poor (only ‘function’ etc. without further details).

Appendix: Updates with respect to version 1.4

Version 1.5 differs in many details from its predecessor, version 1.4b, that was distributed in June 1988 at Madrid. The overall execution is much faster (by a factor of 20 for some examples). This was achieved by changing the handling of recursive values.

In version 1.4, all system files had to be loaded into the working directory, and the files ‘tr...’ were created in the directory ‘/tmp’. This was changed since a ‘mv’ from ‘/tmp’ to the working directory - this was sometimes done by the interpreter - involves a copy and takes much more time than a ‘mv’ within the working directory.

In version 1.4, there were more possibilities to process the results of a TrafoLa command. They were dropped since the user input was not saved in the log file, but could influence the binding performed by the command. In addition, they required that the user was always consulted after the output of the last result, and even after the output of the final ‘No’.