LOGIC

By: Roberto Ghislanzoni ("Roberto_Ghislanzoni".MKTRXI@Xerox.com)

Uses: TABLEBROWSER

This document last edited on 20-Dec-1988 00:52:22.

NOTES ABOUT LOGIC MEDLEY RELEASE

This LOGIC release(1.3) is more robust than previous on top of Lyric; there are some bugs fixed. The major enhancement is the possibility of handle multiple SEDIT sessions: there is a global variable, *LOGIC-CLOSE-ON-COMPLETION-FLG* (defaulting to T), that controls the behaviour of the editing: NIL means not to close the editing window and not to perform a check on the correctness of the axiom just typed in, T means to check definitions and to close the editing window.

INTRODUCTION

This package is devoted to people who want to use a logic paradigm in their programming environment. LOGIC was initially developed in Franz Lisp at the Computer Science Department of the University of Milan: now a modified part of its kernel is running in Common Lisp, so it is possible to use it under every machine running CL: within the Xerox environment, some features are available in order to ease the construction of the programs.

All of the source codes are available: sorry if they are awful! But it's better to have efficiency than syntactic sugar ...

LOGIC MANUAL

LOGIC is essentially a theorem prover based on Horn clauses: the user is allowed to create many theories and within these theories to specify some predicates (clauses); as FOL does, it is also possible to specify some *semantic attachements* (SA), in order to use all the capabilities of the environment: in our implementation, these SAs are expressed in Lisp. A goal proof is performed within specifed theory(es); the user is allowed to dinamically change the theories involved.
These are the elements of the language:
• a *variable* is an atom beginning with '?'
• an *atomic formula* is a list beginning with the name of the predicate and followed by the terms: (on table book)
(mother ?x ?y)
• a *clause* is a list beginning with the consequent, followed by the special symbol ':-', and by the sequence of the antecedents:

((grandfather ?x ?y) :- (father ?x ?z) (father ?z ?y))
• a *set of clauses* is the definition of a predicate
(((append () ?a ?a)) ((append (?a . ?b) ?c (?a . ?d)) :- (append ?b ?c ?d)))
• a *theory* is a set of the definitions of some predicates

HOW TO LOAD AND INIT LOGIC

In order to use LOGIC, load the files LOGIC and LOGIC-UNIFIER. From within the Xerox Lisp Environment, you can also load the development environment LOGIC-DEVEL.DFASL. After loading the files, call the functions:

• (CREATE-BACKGROUND-THEORY): this function creates the main theory reading the data it needs from the file LOGIC.LGC.

• (CREATE-VARIABLES): this functions creates and initializes the variables used by the unifier; it takes a few time to perform its job. This call is due to a lack of the Xerox garbage collector: since the unifier uses techniques of redenomination, a great number of symbols is generated; the 1186 does not release the space used by these symbols, and so they fill up the GC table. The workaround is to re-use all the symbols generated.

If you want to port this code on another machine running CL, it is a matter of taste to eliminate this piece of code, with a little hacking on the source codes.

These are the functions available from the top-level executive of Lisp:

(ALL *VARS CONJ THS*) [Function]

Returns the *vars* that satisfies the goal (*conj*) in the list of theories (*ths*): the background theory is always used.For example you can ask the system to prove:
(ALL '(?a ?b) '((append ?a ?b (1 2 3))) '(append-theory))
--> ((NIL (1 2 3)) ((1) (2 3)) ((1 2) (3)) ((1 2 3) NIL))

(ANY *HOW-MANY VARS CONJ THS*) [Function]

Returns *how-many vars* that satisfies the goal:
(any 2 '?a '((append ?a ?b (1 2 3))) '(append-theory))
--> (NIL (1))

(ATTACH *SA-NAME DEFINITION THEORY-NAME*) [Function]

Allows to create semantic attachments:
(ATTACH 'createw '(lambda (name) (IL:CREATEW () name)) 'my-theory)
and now:
(ANY 1 () '((createw "Kiss me on my lips")) '(my-theory)

--> ;;creates a window on the screen

(CREATE-THEORY *THEORY-NAME*) [Function]

Creates a brand-new theory with that name: return the name of the theory created, not the theory itself.

(LIST-ALL-THEORIES) [Function]

Return a list of the defined theories, currently available.

(LOAD-THEORY *THEORY-NAME*) [Function]

Loads from the current directory the specified *theory-name*; the name of the theory file has the extension .LGC , and it must be previously created by the corresponding function SAVE-THEORY

(LOGIC-ADDA *PRED CLAUSES THEORY-NAME*) [Function]

Adds to the definitions of the predicate *pred* the specified *clauses*, that holds in the theory *theory-name*: the clauses are put in front of the already existing clauses:
(LOGIC-ADDA 'C '(((C 1)) ((C ?x) :- (A ?x))) 'my-theory)

(LOGIC-ADDZ *PRED CLAUSES THEORY-NAME*) [Function]

Adds to the definitions of the predicate *pred* the specified *clauses*, that holds in the theory *theory-name*: the clauses are put at the end of the already existing clauses:
(LOGIC-ADDZ 'C '(((C 2)) ((C ?x) :- (A ?x) (B :y))) 'my-theory)

(LOGIC-ASSERT *PRED CLAUSES THEORY-NAME*) [Function]

Replaces all previous definitions of the predicate *pred* with *clauses*.

(LOGIC-DELETE *PRED-OR-SA THEORY-NAME*) [Function]

Erases from the theory *theory-name* the definition of *pred-or-sa*, that may be either a predicate or a semantic attachment

(LOGIC-DELETE-FACT *FACT-NAME FACT-CLAUSE THEORY-NAME*) [Function]

Erases from the definition of the clauses on the predicate *FACT-NAME* the specified clause *FACT-CLAUSE*, within the theory *THEORY-NAME*.

(MERGE-THEORIES *NEW-THEORY-NAME &REST LIST-OF-THEORIES)* [Function]

Creates the new theory *NEW-THEORY-NAME* made up by all the predicates and sas that hold in all the theories *LIST-OF-THEORIES*: now no control is performed on the consistency in the merging of the theories

(PROVE *CONJ THS*) [Function]

Calls the prover on the specified goals *conj*. *THS* is a list of the theory(es) used. PROVE returns only T or NIL

(SAVE-THEORY *THEORY-NAME*) [Function]

Writes on the local directory the contents of the theory *theory-name*. You will find later a file whose name is composed by the theory name and the extension LGC.
The format of the contents of the file is the following:

```
theory-name
number of semantic attachments
<sa name1> <sa definition>
..
<sa nameN> <sa definition>
number of predicates
<predicate name 1> <clauses for predicate 1>
..
<predicate name N> <clauses for predicate N>
```

A theory file (with .LGC extension) may be created by the user employing a text editor like Emacs or VI (on Symbolics, SUNs etc.), avoiding the saving of the theory every change he performs.

(SHOW-DEFINITION *ELEMENT THEORY-NAME*) [Function]

Shows the definition of *element*, either a predicate or a semantic attachment.

(SHOW-THEORY *THEORY-NAME &OPTIONAL VERBOSE*) [Function]

Shows the contents (name of predicates and sas) of the theory *theory-name*; if *verbose* is T, all the definitions are shown.

THE BACKGROUND THEORY

In the background theory, many interesting primitive predicates are available:

! [Predicate]

The cut predicate, well-known to the PROLOG programmers: a tipical example of its use can be the definition of the predicate NOT:
(((not ?formula) :- (wff ?formula) ! (fail))
((not ?formula)))

(TRUE) [Predicate]

This predicate always succeds

(FAIL) [Predicate]

The predicate that never succeds

(PRINT ?arg) [Predicate]

Prints the argument ?arg passed by

(EVAL&PRINT ?arg) [Predicate]

This predicate evaluates and print the result of evaluation of the form ?arg:
(prove '((eval&print (+ 3 4))) '(my-theory))
7
T

(LOGIC-ADDA *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]

Adds in front of the clauses that define the predicate ?PREDICATE-NAME in the theory ?theory-name the other clauses ?CLAUSES

(LOGIC-ADDZ *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]

Adds to the end of the clauses that define the predicate ?PREDICATE-NAME in the theory ?theory-name the other clauses ?CLAUSES

(LOGIC-ASSERT *?PREDICATE-NAME ?CLAUSES ?THEORY-NAME*) [Predicate]

Replaces all definition for the predicate ?PREDICATE-NAME in the theory ?THEORY-NAME with the new clauses ?CLAUSES

(LOGIC-DELETE *?PREDICATE-OR-SA--NAME ?THEORY-NAME*) [Predicate]

Deletes all definition for predicate (or sa) ?PREDICATE-OR-SA-NAME in the theory ?THEORY-NAME

(LOGIC-DELETE-FACT *?FACT-NAME ?FACT-CLAUSE ?THEORY-NAME*) [Function]

Erases from the definition of the clauses on the predicate *FACT-NAME* the specified clause *FACT-CLAUSE*, within the theory *THEORY-NAME*.

(SET ?var value) [Predicate]

With this predicate it is possible to set a variable within the demonstration (remind that a variable always starts with a '?'):
(prove '((set ?x (list 'a 'b 3))(print ?x)) '(my-theory))
--> (a b 3)
T

(RETRACT ?theory-name) [Predicate]

Tells the interpreter that it must use no more the theory *?theory-name* during the ongoing demonstration; this elision is made only on the current active node of the demonstration tree

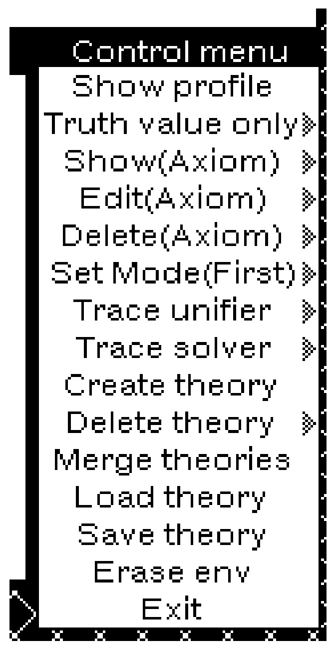
(USE-THEORY ?theory-name) [Predicate]

Tells to the interpreter that it must use the theory *?theory-name* for the ongoing demonstration.

(WFF ?formula) [Predicate]

This is a second order predicate that allows you to prove the truth value of the well formed formula *?formula*

If you load only the LOGIC files, this is the environment you have. On Xerox machines, you can also load the file LOGIC-DEVEL, that allows you to have the development environment: a new entry in your background menu is created, and so you are able to open a logic demonstration window.
This is the control menu:



SHOW-PROFILE: shows the current profile: the MODE of demonstration (FIRST, ALL, INTERACTIVE), and the tracing flags on unifier and solver

TRUTH VALUE ONLY: this flag controls if the prover returns all the goals with the variables instantiated or only the values T or NIL

SHOW AXIOM: shows the definition of an axiom or of a semantic attachment

EDIT AXIOM: edits the definition of an axiom or of a semantic attachment

DELETE AXIOM: deletes the definition of an axiom or of a semantic attachment

SET MODE: sets the mode of the demonstration: this may be ALL, FIRST or INTERACTIVE

TRACE SOLVER: the solver is the procedure of the interpreter that takes as arguments a tree, a formula and the clauses for that formula, and gives back the new tree obtained by the resolution operation; its behaviour is traced on a debugging window which has the middle menu capability of dribbling; the output file has the extension TRC.

TRACE UNIFIER: traces the going on of the unifier on a debugger window; this window too has the middle menu capability of dribbling its output. The pattern, the datum and the unification environment will be shown to the user.

CREATE THEORY: creates a new theory

DELETE THEORY: all the theories loaded are showed in a tablebrowser at the left of the main window; when you select one or more theories, this means that you want to use them for your demonstration; this command deletes the selected theories; you can however undelete or expunge them with the subitems of this entry. Remember that, for undeleting the selected theories with the tablebrowser mark(▶), you must click middle button on it and press CTRL (PROP) key

MERGE THEORIES: merges the selected theories in a new theory; the user is prompted for the name of the new theory

LOAD THEORY: loads a theory from a file in the current directory

SAVE THEORY: save the selected theory(ies) on the corresponding files

ERASE ENV: erases all the environment of the window

EXIT: exits from the environment

Remember that, for every demonstration requested, there must be at least one theory selected in the tablebrowser at the left of the main window

I hope these notes help you to use LOGIC.

You can find some examples in the theory file LOGIC- EXAMPLES.LGC.

Any suggestion is welcome: since it is not fully tested, please notify every kind of error or bug you will find.

EXAMPLES

Choose LOGIC from the background menu: a new window will appear: choose LOAD THEORY from the control menu and type in LOGIC-EXAMPLES at the request in the prompt window: mark the theory loaded in the theories window and try:

((APPEND ?A ?B (1 2 3)))

the system will respond you

((APPEND NIL (1 2 3) (1 2 3)))

Click now on SHOW PROFILE: you will see

MODE: FIRST /Unifier: NOTRACE /Solver: NOTRACE /Values: NIL

Choose SET MODE ALL (submenu) and retry the same goal as before: you get the answer:

((APPEND NIL (1 2 3) (1 2 3)))

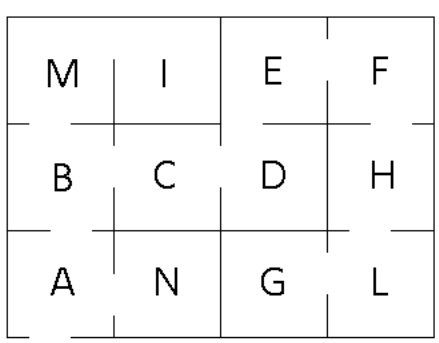
((APPEND (1) (2 3) (1 2 3)))

((APPEND (1 2) (3) (1 2 3)))

((APPEND (1 2 3) NIL (1 2 3)))

NIL

In the theory LOGIC- EXAMPLESa simple little maze is described: type in the goal:
((search a g))
that will find a path from the room 'a' to the room 'g'.



Here are other examples of goals you can try:
((sa-member 3 (1 2 3 4 5)))
((logic-member 3 (1 2 3 4 5)))
The first one is a SA, the latter is a predicate.
((NOT (A 1))) --> T

((NOTMEMBER 2 (1 3 4))) --> T
and so on.
Try now all the other features of the language.
You can ask the system for the same goals from the lisp listener:
(load-theory 'logic-examples)
(prove '((APPEND ?X ?Y (1 2 3 4)) '(logic-examples)) --> T
(all '(?X ?Y) '((APPEND ?X ?Y (1 2 3 4)) '(logic-examples))
--> ((NIL (1 2 3 4)) ((1) (2 3 4)) ((1 2) (3 4)) ((1 2 3) (4)) ((1 2 3 4) NIL))

Have fun!