User's Guide to the MVL System

Matthew L. Ginsberg

Department of Computer Science
Stanford University
Stanford, California 94305

Updated June 8, 1992

Contents

1 Introduction

1.1 What you got

The MVL system is currently distributed via anonymous ftp from t.stanford.edu. A .DVI version of this file, the MVL manual, can be found in the **papers** directory on this machine.

The `mvl` directory contains the source code for the MVL system itself, which is written in standard Common Lisp. It should be possible to port the system to any Common Lisp without substantial difficulty. In addition, the **papers** directory contains .DVI versions of the following papers that relate to the functioning of the system:

1. "Multivalued logics: A uniform approach to inference in artificial intelligence." This paper gives a formal description of multivalued logics and the MVL system.
2. "Bilattices and modal operators." This paper describes the formal connection between the MVL system and modal operators.
3. "Modalities and interrupts." This paper describes the formal mechanism that exists for interrupting the theorem prover before its analysis of a query is complete.
4. "Computational considerations in reasoning about action" and "Computational considerations in reasoning about action II: Planning." These papers describe the application of MVL to planning problems.

The MVL sources are public domain. You are free to use, modify, and distribute them as you see fit, provided that such use or distribution is not for commercial purposes. No warranties are made with regard to the performance of the system, however, and no commitment of support has been made.

1.2 Overview

This user's guide has been written to help you install and use the MVL system. It describes the basic structure of the system, and shows you how to take advantage of the flexibility of multivalued inference.

Essentially, MVL is just like any other logic programming system. There is a database consisting of those sentences about which the system knows something, and there are facilities for determining whether or not some query follows from the information the database contains. Viewed in this fashion, the MVL system is not much different from a conventional logic programming language such as PROLOG.

What makes MVL unique is that each sentence in the database is labelled with a *truth value*. Thus, instead of simply assuming a statement to be true and dumping it into the database, we will *label* the statement as true when we insert it. Of course, we can use other labels as well: true by default, true by virtue of some assumption, or even false or false by default. MVL considers these truth values when determining how to respond to a PROLOG-like query.

This manual is divided into two parts. In the first part (Sections 3 – 11), we describe the use of the system where the truth values assigned to sentences are simply "true" or "false." Used in this fashion, MVL is not much different from PROLOG.

The second part of the manual shows you how to use logics involving more complex truth values. We examine the logics that have been predefined as part of the MVL system, and describe the various ways in which you can define your own logics. Several tools for this purpose have been included as part of the MVL system. In addition, we describe MVL's facilities for handling modal operators, which are operators that accept logical sentences as arguments.

1.3 Why MVL is unique

As mentioned in the previous section, what distinguishes MVL from other declarative programming languages is its support for multiple truth values. There are other differences, however, and these will be discussed in detail in subsequent sections. Here's a quick summary:

1. As mentioned, the principal feature offered by MVL is its support of multiple truth values.

2. The basic theorem prover used by the system includes facilities to detect loops and control recursion encountered in proof search. The first-order theorem prover is discussed in Section 7, although we will have nothing further to say about recursion control.

3. The system can handle negative subgoals containing free variables as discussed in Section 4.2.

4. The MVL unifier includes an occurcheck and full support for sequence variables. This is discussed in Section 3.1.1.

2 Installation and testing

The MVL system is currently developed and maintained on a Sparcstation running Allegro Common Lisp and is tested only infrequently on other platforms. Although the system has been written in standard Common Lisp to maximize ease of portability and no problems have been encountered when moving the system to a variety of platforms, performance on other systems is not guaranteed.

MVL currently supports both ANSI Common Lisp and Common Lisp as described in Steele's 1984 book. In order to determine whether or not the ANSI standard is conformed to, the system checks to see if `in-package` is a macro or a function. If the former, it is assumed that the entire ANSI standard is supported; if the latter, it is assumed that the language is as in Steele and various extensions are defined. It is expected that some future release of MVL will support the ANSI standard only.

2.1 Sun-4

In order to load the system on a Sparcstation running Allegro Common Lisp, do the following:

1. (ANSI) Set up a logical pathname translation so that logical pathnames beginning with `MVL` point to the directory where MVL is kept. (non-ANSI) Edit the file `mvldcl.lisp` so that the `:default-pathname` field in the MVL system definition points to the directory where MVL will be kept. This directory should also contain the following files:
   (a) `f1.mvl, recursion.mvl, def-test.mvl, circ-test-1.mvl, circ-test-2.mvl, circ-test-3.mvl, circ-test-4.mvl, completed-db-1.mvl, completed-db-2.mvl, modal-test.mvl, actions.mvl, shooting.mvl, plan.mvl, blocks.mvl`
   (b) `mvl.test, bc.test, recursion.test, atms.test, defaults.test, circumscription.test, completed-db.test, modal.test, fc.test, temporal.test, plan.test`
   (c) `mvl.demo, bc.demo, atms.demo, defaults.demo, circumscription.demo, completed-db.demo, modal.demo, fc.demo, temporal.demo, plan.demo`

2. Compile and load the file `mvldcl.lisp`.

3. Execute the function `(mvl:compile-mvl)` to compile and load the MVL system. Using this function in the future will only compile those files that have been changed since last compiled.

4. Invoke the function `(use-package 'mvl)` to access the functions and symbols of the MVL system (which is kept in its own package; see Section 11.2).

Now invoke the function `(test-mvl)`, which should return 0 (i.e., 0 errors encountered). If it doesn't, the problem is probably that MVL is unable to locate the files mentioned in (1) above.

2.2 Symbolics 3600 series

The MVL system will run on Symbolics Genera 7.2 or later. It is possible to run the system using Genera 7.0 or 7.1, although a bug in the implementation of the Common Lisp `&allow-other-keys` keyword may cause the system to dump to the cold load stream if certain predicates are procedurally attached. (See Section 9.)

You should begin by editing the file `mvldcl.lisp` so that the default pathname in the MVL system declaration points to the directory in which the files are actually kept. You can then load the MVL system using either the Load System command or the Lisp function `(mvl:load-mvl)` that is defined in the file `mvldcl.lisp`. Now access the MVL package using `(use-package 'mvl)` and execute the function `test-mvl`. This function should return 0 (i.e., 0 errors encountered). If it doesn't, the problem is probably that MVL is unable to find the directory its test files are on.

2.3 Other machines

The MVL system has also been run and tested on a wide variety of other machines. In order to load the system, follow the instructions provided for Sun-4's.

3 The MVL database

3.1 Representation

3.1.1 Structure of the database

As already remarked, the MVL database consists of sentences labelled with truth values. In this section, we will describe the form of the sentences themselves; in the next section, we comment briefly on the truth values used to label them.

Sentences in the database are represented as LISP s-expressions using the prenex normal form of the standard logical connectives `NOT`, `OR` and `AND`. Thus the statement that Fred is tall and Harry is not would be written as:

```
(AND (TALL FRED) (NOT (TALL HARRY)))
```

Any LISP atom can be used as a constant term, with the exception of variables (see below) and the following reserved connectives:

```
NOT AND OR => <= <=> IF IFF
```

The connective <= is used in backward-chaining rules;

```
(<= CONCLUSION PREMISE-1 ... PREMISE-N)
```

is logically equivalent to

```
(IF (AND PREMISE-1 ... PREMISE-N) CONCLUSION)
```

as is the forward-chaining version

```
(=> PREMISE-1 ... PREMISE-N CONCLUSION)
```

Although logically equivalent, these expressions behave differently in practice. The version

```
(IF (AND PREMISE-1 ... PREMISE-N) CONCLUSION)
```

in which the connective `IF` is used represents a pair of rules, depending on the value of the parameter `*if-translation*`. If the value is `bc` (the default), then two backward-chaining rules are produced, so that

```
(IF (A) (B))
```

is equivalent to

```
(AND (<= (B) (A)) (<= (NOT (A)) (NOT (B))))
```

the conjunction of the original rule and its contraposition. If the value of `*if-translation*` is `fc`, then forward-chaining rules are produced and the translation is to

```
(AND (=> (A) (B)) (=> (NOT (B)) (NOT (A))))
```

Finally, if `*if-translation*` has the value `mix`, the pair of noncontraposed rules

```
(AND (<= (B) (A)) (=> (A) (B)))
```

is produced.

Equivalences are represented using either `IFF` or `<=>`. Thus

```
(IFF (FOO FRED) (GOO FRED))
```

is equivalent to

```
(AND (IF (FOO FRED) (GOO FRED)) (IF (GOO FRED) (FOO FRED)))
```

Translations involving `<=>` depend on the value of the parameter `*equivalence-translation*`. As above, if the value is `bc` (the default), then two backward-chaining rules are produced, so that

```
(<=> (FOO FRED) (GOO FRED))
```

translates to

```
(AND (<= (FOO FRED) (GOO FRED)) (<= (GOO FRED) (FOO FRED)))
```

If `*equivalence-translation*` has the value `fc`, two forward-chaining rules are produced so that the translation is to

```
(AND (=> (FOO FRED) (GOO FRED)) (=> (GOO FRED) (FOO FRED)))
```

Finally, if the value is `mix`, one forward-chaining rule and one backward-chaining rule are produced:

```
(AND (=> (FOO FRED) (GOO FRED)) (<= (FOO FRED) (GOO FRED)))
```

Variables are represented using LISP atoms beginning with the character `?` and are handled as in PROLOG, so that free variables in database statements are implicitly universally quantified. Again as in PROLOG, existential quantification is handled via Skolemization; thus the backward-chaining sentence "Every rich person is happy" and the sentence "There is a rich person" would be written as

```
(<= (HAPPY ?P) (PERSON ?P) (RICH ?P))          (1)
```

and

```
(AND (PERSON P0017) (RICH P0017))
```

respectively.

The variables in database sentences are standardized apart from the other variables used by the system. This is done by using uninterned variables in the actual representation of a database fact, so that the internal representation of (1) would be

```
(<= (HAPPY #:?P) (PERSON #:?P) (RICH #:?P))
```

The original variables are generally reinstated when MVL is asked to display a database proposition.

3.1.2 Sequence variables

In addition to the simple variables discussed in the previous section, MVL supports the use of "sequence variables." These variables begin with the characters `?*`, and will match not a single atom, but an arbitrary (possibly empty) sequence of atoms.

Thus, for example,

```
(FOO A B)
```

will unify with

```
(FOO ?*X),
```

returning the binding list

```
((?*X . (A B)))
```

in which `?*X` has been bound to the list `(A B)`. This binding list will presumably be displayed as

```
((?*X A B)).
```

Similarly, the most general unifier of `(FOO)` and `(FOO ?*X)` is given by `((?*X))`.

The presence of sequence variables may cause the most general unifier of two patterns to be nonunique. For example, the result of unifying

```
(FOO A B)
```

with

```
(FOO ?*A ?*B)
```

can be any one of the three answers

```
((?*A A B) (?*B))
((?*A A) (?*B B))
((?*A) (?*B A B))
```

For this reason, the unification functions return not a single answer, but a *list* of possible unifiers.

3.1.3 Functions to manipulate database sentences

The following functions and global parameters are used to manipulate sentences in the database:

`varp(x)` Returns `T` if x is a variable, and `NIL` otherwise.

`varp?(x)` Returns `T` if x is a standard (i.e., non-sequence) variable, and `NIL` otherwise.

`varp*(x)` Returns `T` if x is a sequence variable, and `NIL` otherwise.

`vartype(x)` Returns ? if x is a standard variable, $*$ if x is a sequence variable, and `NIL` is x is not a variable.

`new-?var()` Returns a new temporary variable.

`new-*var()` Returns a new temporary sequence variable.

`normal-form(p)` Returns an equivalent form of p suitable for insertion into the database (i.e., with the connectives `IFF`, `IF` and <=> reduced to <= and =>).

`*if-translation*` Controls the fashion in which `normal-form` deals with the connective `if`.

`*equivalence-translation*` Controls the fashion in which `normal-form` deals with the connective <=>.

`negate(p)` Returns the negation of p.

`standardize-operators(p)` Returns an equivalent form of p with the connectives `<=>`, `IF` and `IFF` removed. (Note: This function does not use the control variables `*if-translation*` and `*equivalence--translation*`, since it is used when proving sentences, not when inserting them into the database.)

`cnf(p)` Returns p in conjunctive normal form, as a list of conjuncts, where each conjunct is a list of disjuncts. Thus

```
(CNF '(AND A B))
```

returns

```
((A) (B))
```

`dnf(p)` Returns p in disjunctive normal form, as a list of disjuncts, where each disjunct is a list of conjuncts. Thus

```
(DNF '(AND A B))
```

returns

```
((A B))
```

3.2 Truth values

The truth values used to label the sentences in the MVL database are elements of a bilattice that can be selected by the user with the function `load-logic`. We will defer a discussion of these truth values to the second portion of this manual, which discusses the use of bilattices that label sentences as other than simply true or false. At this point, we merely remark that if it is our intention to assert the sentence `(not (foo))`, this is actually accomplished by asserting that the negated sentence `(foo)` has truth value `false`.

3.3 Database form and internals

The database is maintained using a discrimination net, as suggested by Charniak et. al in their book on LISP programming. Any particular sentence in the database, say `(tall tom)`, is associated with a LISP atom such as `P27` that is its "proposition number." The propositions are numbered consecutively, beginning with `P1`. At any given point in time, the function `(props)` can be invoked to return a list of all of the propositions in the database.

Very few of the functions that are used to manipulate the internal form of the database are likely to be needed by a user of the MVL system. The following are the most useful:

`props()` This function returns a list of all propositions about which any information has been recorded.

`denotes(a)` Returns the proposition associated with a particular LISP atom.

`truth-value(a)` Returns the truth value associated with a particular LISP atom. Thus, if the result of asserting `(not (short tom))` into the database was the atom `P38`, `(denotes 'p38)` would evaluate to `(short tom)`, and `(truth-value 'p38)` would evaluate to `false`.

`index(p)` Updates the discrimination net as needed to insert the proposition p into the database.

`unindex(p)` Removes from the discrimination net information relating to the proposition p.

4 Quantification

As we have remarked, quantification is treated in MVL in a fashion similar to that used by PROLOG systems: Free variables appearing in the database are assumed to be universally quantified; free variables appearing in queries are existentially quantified. Skolemization is used to handle existential quantification in database statements or universal quantification in queries.

The use of this approach means that the answer to an existentially quantified query will often be of a form that indicates how the variables should be bound in the result. In MVL these binding lists are represented as expressions of the form

```
((var-1 . bdg-1) ... (var-n . bdg-n))
```

where `var-1` through `var-n` are variables and `bdg-1` through `bdg-n` are their respective bindings. If no variable is bound to a value, the empty binding list `NIL` is used.

As an example, suppose that the database contains the single sentence

```
(LIKES COCO ?X)
```

indicating that Coco likes everything. Now the query

```
(LIKES ?Y MATT)
```

will be interpreted as, "Find something that likes Matt." The answer returned will be

```
((?Y . COCO))
```

since Coco likes Matt. Since the queries

```
(LIKES COCO MATT)
```

and

```
(LIKES COCO ?Z)
```

both succeed without binding a variable, the answer `NIL` will be returned for each.[1]

4.1 Functions used to manipulate binding lists

A variety of functions within MVL are used to construct and manipulate binding lists. Here are some of the more useful ones:

`groundp(p)` Returns `T` if the proposition p is ground (i.e., contains no variables) and `NIL` otherwise.

`vars-in(p)` Returns a list of the variables appearing in p.

`get-bdg(var bdg-list)` Returns the value assigned to the variable `var` by the binding list `bdg-list`. If the variable does not appear in the binding list, `NIL` is returned.

`delete-bdg(var bdg-list)` Destructively modifies `bdg-list`, removing any binding for the variable `var`.

`plug(p bdg-list)` Returns the result of instantiating the proposition p using the given binding list. This function is nondestructive.

`samep(p1 p2)` Returns `T` if the propositions p_1 and p_2 are identical (up to variable renaming), and `NIL` otherwise. If the optional `return-answer?` argument is supplied and the propositions are the same, a binding list is returned instead of `T`.

`unifyp(p1 p2)` Returns a list of the most general unifiers of p_1 and p_2, or `NIL` if the propositions fail to unify.

`instp(p1 p2)` If p_2 is an instance of p_1, return a list of their most general unifiers; only variables in p_1 are bound. Returns `NIL` if p_2 is not an instance of p_1.

[1] In order to distinguish the empty binding list `NIL` from a failed database query, all queries return a binding list as well as a truth value. This is discussed in Section 6.1.

`matchp(pattern atom)` Returns a list of elements of the form `(b . d)` where b is a most general binding list such that `(plug pattern b)` is an instance of the proposition labelled with `atom` and d is a binding list for the variables in the database proposition. `NIL` is returned if no such binding list exists. Thus if `P1` is `(TALL #:?2)`, and `P2` is `(TALL TOM)`, then

```
(matchp (TALL TOM) P1)
```

returns a list consisting of the single entry

```
(NIL . ((#:?2 . TOM)))
```

The `car` of this entry is `NIL`, indicating that no variables in the query need to be bound to match `P1`; the `cdr` is `((#:?2 . TOM))` since the variable `#:?2` in `P1` needs to be bound to `TOM` to match the query.

```
(matchp (TALL ?X) P2)
```

returns `((((?X . TOM)) . NIL))`.

Additional functions are described in Section 14.6.4.

4.2 Negative subgoals containing free variables

Consider the following PROLOG fragment:

```
f(x) :- not(g(x)).
g(A).
```

The interpretation we will assign to the implication `f(x) :- not(g(x))` will be that it is universally quantified over x, and therefore corresponds to the implication

$$\forall x.[\neg g(x) \supset f(x)]$$

Given this interpretation, what should the response be if we provide the system with the query `f(x)`?

In light of the negation as failure rule of PROLOG (which has analogs in the MVL system), we would expect the query to succeed for all values of x *except* $x = A$.

Most PROLOG systems cannot respond to queries such as this one, since a nonground negative subgoal is encountered during the proof process. One approach to these queries, suggested by David Chan, is to do just what we have suggested in the previous paragraph, computing those binding lists for which the query fails.

Unfortunately, there are many queries that succeed in general, but for which it is impractical to compute all of the bindings for which the query fails. This may be because there are infinitely many such failures, or because the time needed to compute them explicitly is prohibitive. Situations such as this arise frequently in applications of declarative programming techniques to planning problems; here, we would like simply to return an indication that the query "succeeds in general." In the above example, we might report that the query had succeeded without binding the variable x; even though $x = A$ is an exception, the report of success is true in general.

Both approaches are supported by MVL, and the global variable `*completed-database*` is used to distinguish between them. If this variable has the value `T`, all exceptions are computed; if it has the value `NIL`, the other approach is taken. The default value is `NIL`:

`*completed-database*` If `T`, find all exceptions to default proofs. If `NIL` (the default), accept defaults proofs with possible exceptions as complete.

5 Modifying the MVL database

5.1 Keyboard input

The simplest functions to modify the MVL database simply accept propositions from the user and either add or remove them from the knowledge base. The primitive function for adding to the database is `stash`; the function `unstash` is used for removal.

The function `stash` accepts a proposition and up to three keywords:

`value` This keyword is used to supply a specific truth value to be assigned to the proposition. If no value is supplied, the default value is used (this value is supplied when the logic is chosen; see Section 12).

`increment-p` If the proposition in question already appears in the database, the user has a choice: The given truth value can be used to replace the one currently appearing in the database, or it can be combined with the existing value using the combination function supplied with the logic being used. The truth value is replaced if `increment-p` is `NIL`, and is updated if it is `T` (the default).

`allow-procedural-attachment` This keyword, which defaults to `T`, controls whether or not it is possible to procedurally attach the action of stashing the supplied proposition. See Section 9.

Somewhat more useful than `stash` is the function `state`. This function is identical to `stash`, except it translates the supplied sentence into a form suitable for inference by using the function `normal-form` described in Section 3.1.3. In addition, `state` cannot be procedurally attached, so the keyword `allow--procedural-attachment` is not allowed.

For removing sentences from the database, the functions `unstash` and `unstate` are used.

The function `unstash` accepts a proposition to be removed from the database, and up to three keywords: `value`, `decrement-p` and `allow-procedural-attachment`. The last of these is treated as for `stash`. In general, if no keywords are used, the effect is to remove the proposition from the database.

If `decrement-p` is `NIL`, then `unstash` behaves like `stash`, except the default value used for unstashing is not necessarily the same as the default value used for stashing.

If `decrement-p` is `T`, then the new truth value assigned to the proposition is given by

$$o \cdot \neg v \tag{2}$$

where o is the old value and v is the value supplied via the `value` keyword. The $\cdot$ and $\neg$ operations are as described in Section 12.

There is one additional function for removing sentences from the database, `empty`. This function accepts no arguments, but does take a single keyword, `value`. The function calls `unstash` for every sentence in the database.

All of the functions described in this section work by calling the primitive function `new-val`. This function accepts as arguments a proposition name and a truth value, and sets the truth value of the proposition to the given value. If the truth value is `unknown`, indicating that nothing is known about the proposition, it is removed from the database and the discrimination net.

5.2 Input from files

It is also possible to enter a database from a disk file. These files should in general have the extension `.MVL`, and should be located either on the currently active disk area or on the MVL directory. The files should consist simply of the sentences to be added to the database.

In addition to consisting of such sentences, the files may contain the keyword `:value`, followed by the value to be assigned to each sentence, or the keyword `:function`, followed by a function to be applied to each sentence in order to *compute* the value to be assigned to it. (A value of `NIL` indicates that the previous value should no longer be used and the default should be used instead.) The keyword `:empty` empties the database before loading the file. Examples of these constructs can be found in the various `.MVL` files included with MVL.

The sentences are loaded with `mvl-load`, which accepts a file name and an optional keyword `empty`. If the keyword is supplied, the database is emptied before the file is read; in any event, `state` is then applied to each sentence in the given file. The sentences in the file can be removed from the database with the function `mvl-unload`, which accepts a file name as its argument. These functions each call `mvl-file`, which accepts a string and returns a path to the `.MVL` file associated to it.

Related to `mvl-load` is `mvl-save`, which also accepts a file name as an argument, but instead of loading a declarative database from the file, writes the current database *to* the file.

5.3 The editor interface

5.3.1 Sun EMACS editor

In addition to the file commands `mvl-load` and `mvl-unload`, there is an interface to the EMACS editor on Sparcstations. For a file of mode MVL, certain keys are bound as follows (each of these commands also recognizes the `:value` and `:function` keywords):

`meta-S` Applies `state` to each s-expression in the current region.

`meta-U` Applies `unstate` to each s-expression in the current region.

`meta-s` Applies `state` to each s-expression in the current defun (i.e., s-expression beginning in column 1).

`meta-u` Applies `unstate` to each s-expression in the current defun.

In order to use these commands, certain modifications will need to be made to your `.emacs` file. The file `dot.emacs` is an example of what to do.

5.3.2 Symbolics ZMACS editor

In addition to the file commands `mvl-load` and `mvl-unload`, there is an interface to the ZMACS editor on Symbolics 3600 series machines. For a file of mode MVL, certain keys are bound as follows (each of these commands also recognizes the `:value` and `:function` keywords):

`super-s` Applies `state` to each s-expression in the current region.

`super-u` Applies `unstate` to each s-expression in the current region.

`super-f` Applies `fc` to each s-expression in the current region. See Section 8.2.

`super-e` Applies `erase` to each s-expression in the current region. See Section 8.2.

5.4 Summary of database modification functions

`stash(p)` Inserts p into the database. Allowed keywords: `value`, `increment-p` and `allow-procedural--attachment`.

`state(p)` Inserts p into the database after converting it to normal form. Keywords: `value`, `increment-p`.

`unstash(p)` Removes p from the database. Keywords: `value`, `decrement-p`, `allow-procedural-attachment`.

`unstate(p)` Removes p from the database after converting it to normal form. Keywords: `value`, `decrement-p`.

`empty` Unstashes every sentence in the database. Keyword: `value`.

`new-val(a v)` Sets the truth value of the sentence associated with the atom a to v. If v is the truth value `unknown`, the sentence is removed from the database.

`mvl-file(string)` Returns the path for the MVL file associated with the given name. The file name may include an extension, one may be supplied as an optional argument, or `.MVL` will be used by default. There is an optional keyword `no-error` indicating that even if the file cannot be found, the path should be returned.

`mvl-load(file)` Loads the sentences in the given file into the MVL database.

`mvl-unload(file)` Removes the sentences in the given file from the MVL database.

`mvl-save(file)` Writes the MVL database out to the given file. This function also accepts the keywords `cutoffs` and `props`, which are treated as for `contents` (see Section 6.3). Finally, the keyword `if-exists` is passed to the `with-open-file` macro.

6 Querying the MVL database

6.1 MVL responses to queries

Because of the fact that a particular fact may be stored in the database with any of a variety of truth values, MVL responds to a query by returning not simply a binding list (as most PROLOG interpreters do, for example), but by returning a binding list *and a truth value.*

In actuality, what is returned is an instance of the `answer` structure. This structure contains the following fields:

`binding` Bindings for some of the variables in the query.

`value` A truth value. The bound version of the query has this truth value.

The definition of this structure leads Common Lisp to define the access functions `answer-binding` and `answer-value`; in addition, a function `equal-answer` is defined by MVL to test two answers for equality:

`answer-binding(a)` Returns the binding field of the answer `a`.

`answer-value(a)` Returns the value field of the answer `a`.

`equal-answer(a1 a2)` Returns `T` if the two answers `a1` and `a2` are equal, and `NIL` otherwise. The functions used are `binding-equal` and `mvl-eq`, the functions normally used to test binding lists and truth values for equality.

Thus, in response to the query

```
(flies ?x)
```

the response of an answer with binding `((?x . Tweety))` and truth value `dt` would indicate that

```
(flies Tweety)
```

has truth value `dt`. In other words, Tweety's flying is a conclusion that is true by default. (See Section 13.3 for a description of default truth values in the MVL system.)

As a side effect, this allows the user to distinguish a response of `NIL` (indicating that the query has failed) from a response in which the binding is `NIL` (indicating that the query has succeeded without binding any variables).

6.2 Searching for specific sentences

There are two functions that are most commonly used to retrieve facts that have been stored in the database, `lookup` and `lookups`. Each of these functions accepts as arguments a proposition to search for and up to four keywords, and returns two values.

The keywords and their meanings are as follows:

`cutoffs` This keyword is used to give conditions on the sentences being considered as possible matches to the query. Any sentence that is considered as a match must have an associated truth value `v` such that `(test v cutoffs)` evaluates to `T` (see Section 12.2.2). The default value of this keyword is defined by the bilattice in use (see Section 12).

`inv` This keyword defaults to `NIL`. If it is `T`, however, any truth values found should be negated before they are returned.

`allow-procedural-attachment` This is similar to the keyword for `stash`.

`succeed-on-modal-ops` If `T`, then any attempt to look up a proposition involving a modal operator will immediately succeed; if `NIL` (the default), then the result returned will be the result of applying the modal operator to the truth values of the propositions that are its arguments. See Section 15.

The two functions `lookup` and `lookups` are identical except that `lookup` looks for only a single match for the supplied proposition, while `lookups` searches for all such matches. Thus while `lookup` returns two values, `lookups` returns two *lists* of values. Each entry on each of these lists is analogous to the corresponding value returned by `lookup`. The values returned by `(lookup q)` are as follows:

1. First, an `answer` is returned. If this answer contains the binding list `b` and the truth value `v`, then applying the binding list to the query produces a result with the given truth value. In other words, `(plug q b)` is an instantiation of a sentence appearing explicitly in the database with truth value `v`.
2. Second, one of the following three values is returned:
   (a) If the sentence was found by procedural attachment, the atom `attach` is returned.
   (b) If the sentence was found by applying a modal operator, the atom `modal` is returned.
   (c) Under other conditions, the instantiated version of the proposition that resulted in the match is returned.

As already mentioned, `lookups` returns two lists of values.

Here are summaries of `lookup` and `lookups`:

`lookup(q)` Searches the database for a sentence matching the query q. Allowed keywords: `cutoffs`, `inv`, `allow-procedural-attachment`, `succeed-on-modal-ops`. Returns two values: an answer and the source of the match. Returns `NIL` if the query cannot be found in the database.

`lookups(q)` Similar to `lookup`, but looks for *all* sentences matching the query. Returns two lists of values.

6.3 Searching the entire database

In addition to searching for a particular sentence, it is also possible to display large portions of the entire MVL database. The basic functions for doing this are `contents`, which displays the entire database, and `prfacts`, which accepts a LISP atom as an argument and displays all of those facts concerning that atom.

Each of these two functions also accepts a keyword argument `cutoffs` that, as for `lookup`, is used to supply a test that must be passed by the truth value of any database sentence before that sentence is displayed. `contents` also accepts a keyword `props` that defaults to `(props)` (see Section 3.3) and is a list of the sentences to be displayed.

Related to **prfacts** is **unstash-facts-with-atom**, which accepts an atom as an argument and removes from the database any fact concerning that atom. **Unstash-facts-with-atom** accepts the same keywords as **unstash** does.

Here, then, are the functions described in this section:

contents Displays the contents of the database. Keywords: **cutoffs**, **props**.

prfacts(atom) Displays the database facts concerning the given atom. Keyword: **cutoffs**.

unstash-facts-with-atom(atom) Unstashes from the database any facts concerning the given atom. Keywords: **value**, **decrement-p**, **allow-procedural-attachment**.

6.3.1 Command line interface

Sun In addition to the above functions, on Sparcstations, the key ⟨meta⟩-* may be typed to the command line processor running from the editor. The result is that any information in the MVL database concerning the currently open predicate will be displayed to the user.

Symbolics On Symbolics systems, the key ⟨super-shift⟩-A may be typed to the command line processor. The result is that any information in the MVL database concerning the currently open predicate will be displayed to the user. This feature is the MVL analog to the ⟨ctrl-shift⟩-A key, which displays information about the arguments to the currently open function.

7 The first-order prover

As discussed in the technical papers describing MVL, inference in MVL proceeds using a conventional first-order theorem prover; in this section, we will describe this prover and the interface to it. Be warned that the information in this section is unlikely to be used by other than the most sophisticated users of MVL; most readers will want to proceed directly to Section 8.

In general, there will be a variety of invocations to the first-order prover active at any particular time. Each such invocation is associated with a task that the multivalued prover needs to complete, and each such task is a LISP object that is an instance of the **consider** structure.[2]

The multiple invocations of the first-order prover are run in parallel by MVL. To each such invocation is associated an instance of the structure **proof** that is used to record temporary information about the progress of the proof effort. We will discuss first the first-order prover itself, then the construction and termination of the invocations of it, and finally the structure of the multivalued proving tasks.

7.1 Nature

The prover itself is a simple natural-deduction style complete theorem prover. Only inferences against the initial clauses of database statements are allowed; the result of this is that after applying **state** to a sentence of the form

```
(<= CONCLUSION PREMISE-1 ... PREMISE-N)
```

`PREMISE-1` will always be tried first when the prover attempts to prove `CONCLUSION`. In a similar way, stating the forward-chaining rule

```
(=> PREMISE-1 ... PREMISE-N CONCLUSION)
```

will result in a situation where only `PREMISE-1` will trigger forward chaining.

The search of the first-order prover is essentially uncontrolled, although there is a single parameter ***depth-limit*** that bounds the depth to which the proof tree will be examined:

[2] In fact, each task is an instance of a structure that *includes* the `consider` structure. See the file `bc.lisp` for further details.

`*depth-limit*` is the limit beyond which the first-order proof search is abandoned. If `NIL` (the default), unconstrained depth-first search is used.

This parameter may be reset by the user:

`set-depth-limit` This function accepts a single optional argument and sets `*depth-limit*` to the given value. The default is `NIL`.

7.2 Invoking the prover

The prover is invoked using the function `init-prover`. This function accepts a proposition as an argument and creates and returns an instance of `proof` that contains auxiliary information about the proof attempt. Here are the fields in the structure:

`prop` The proposition being proved.

`active-nodes` A list of the nodes that are still under active consideration.

`answers` A list of answers that have been found.

`returned-answers` A list of answers that have already been returned. (The actual structure of this field is more complex and is documented in `bc.lisp`.)

`indexp` **and** `indexn` These entries are used to index the various subtasks generated by the prover.

`depth-limit` The value of `*depth-limit*` used in this proof attempt.

`tasks` A pointer back to the tasks using this invocation of the first-order prover.

`vars` The variables that appear in the proposition being proved.

Each node is an instance of the structure `goal`, which is described in the file `first-order.lisp`.

To continue a proof process, the function `cont-prover` is used. This function accepts as an argument an instance of `proof` that contains the auxiliary information for the proof process being resumed. The value returned by `cont-prover` is a structure describing the results of the proof process itself; `cont-prover` returns as soon as the prover finds a single solution to its query. The value returned is a list of instances of the `bcanswer` structure; this structure includes the fields in the `answer` structure and the following additional one:

`just` This field contains a list of those database sentences that were used in the proof.

If the proof attempt fails, `NIL` is returned.

Here is a summary of the functions described in this section:

`init-prover(p)` This function initializes and returns an instance of `proof` containing auxiliary information about the proof process.

`cont-prover(proof)` Resume the task associated to the auxiliary information in the given instance of the `proof` structure. This function returns a list of `bcanswers` corresponding to the result obtained by the prover, and returns as soon as a solution to the original query is found.

7.3 The multivalued proof tasks

The individual proof attempts generated by a multivalued query are instances of structures that include the structure `consider`. The exact details of this and related structures are unlikely to be of much interest to a user of the MVL system, but we include them here in the interests of completeness. These are the fields in the `consider` structure:

`prop` The proposition being investigated by this task.

`parent` The task for which the current task is a child.

`children` Children of the current task.

`depth` Depth of this task in the overall proof tree.

`truth-value` The current best estimate of the truth value that should be assigned to the proposition being considered.

`paths` The MVL system goes to a great deal of effort to ensure that it does not consider proofs that will not affect the eventual response to a user's query; intermediate information used in this computation is stored in this field. Further details can be found in the file `bc.lisp` and the files to which it points.

When the prover must select a task to work on next, it does so via the following function:

`choose-proof-and-tasks` Returns two values: a proof effort, and one or more tasks that use it. This proof effort is examined next, and any answers found are passed to the given tasks. The current implementation of MVL simply selects the shallowest active task.

8 Inference

8.1 Backward chaining

The information in Section 7 is more technical than most users of MVL are likely to need. What is of much greater interest are those database query tools that are like `lookup` and `lookups` but, instead of simply looking for a fact in the database, attempt to derive it as a consequence of other database information. These functions use those of the previous section as described in the associated technical papers.

The functions accept three sorts of cutoff information, instead of the simple information provided to `lookup` and `lookups`. The first, provided via the `cutoffs` keyword, is used to construct a test to decide whether an answer computed by the prover is good enough to be returned when the proof effort is complete. The second, provided via the `succeeds` keyword, indicates that an answer should be returned as soon as it can be shown that the final answer will pass the given test. The final cutoff, supplied via the `terminate` keyword, indicates that the answer should be returned as soon as there is any indication that the final answer *might* pass the given test.

As an example, suppose that we were trying to find a plan to achieve some goal. We might well be prepared to terminate our search as soon as we found a plan that was *guaranteed* to achieve the goal; if we completed our search without finding such a plan, we might then want to return all plans that we *expected* would achieve the goal. It is to facilitate such possibilities that the backward-chaining functions expect three sets of cutoff information. The default value for `cutoffs` is the standard one associated to the active bilattice (see Section 12); the default choice for `succeeds` corresponds to complete information for the singular versions of the queries and guarantees failure for the plural versions (since an answer might be about to be discovered for a different instantiation of the query). The default value for `terminate` always fails.

The backward-chaining functions are `bc` and `bcs`. They return two values: an answer (or list of answers for the plural versions `bcs`) and an instance of the `analysis` structure describing tasks remaining to be considered by the prover. In addition, these functions all accept an optional `analysis` keyword that, if supplied, should be an instance of the `analysis` structure that is to be continued instead of a new one being created. This is to allow the user to continue proof efforts that have returned for some reason.

`bc(q)` Attempts to prove the query q from information in the database, looking only for a single answer.

`bcs(q)` Attempts to prove the query q from information in the database, looking for all possible answers.

`Bc` and `bcs` work by maintaining an instance of the `analysis` structure that includes a variety of information about the status of the current proof attempt. Here are the fields in this structure:

`prop` The proposition being considered by the prover.

`cutoffs` The cutoff information that must be satisfied by any result before it is to be returned.

`succeeds` The cutoff information that will allow the prover to terminate early if an answer is guaranteed to pass the test.

`succeeds-with-bound` If `T` (the default), don't return as soon as you know that an answer will pass the `succeeds` test – only return when a specific lower bound has been established on the answer's truth value. If `NIL`, return as soon as it is known that the answer will pass the success test.

`terminate` The cutoff information that will allow the prover to terminate early if an answer might pass the test.

`active-tasks` A list of the proof tasks (see Section 7.3) that still need to be investigated.

`main-task` The task that was created upon the original invocation of the prover.

`prototasks` A list of tasks that have not yet been initialized by creating proof efforts for them.

`proofs` A list of all proof efforts used.

`returned-answers` A list of answers that have already been returned to the user; this is to keep MVL from compulsively re-returning these answers if the `analysis` is reinvoked.

`current-value` The current best estimate of the truth value that will be returned in response to the overall query.

The top-level functions using this structure are as follows:

`init-analysis(prop)` The MVL primitive upon which the proving functions are based, this function accepts a proposition and creates and returns an instance of the `analysis` structure.

`cont-analysis(analysis)` This function continues the investigation of a specific analysis, and can be used to resume a proof effort that has been curtailed for some reason. It returns two values: an instance of the `answer` structure giving the results of the proof attempts, and a modified version of the `analysis` structure that can be used to continue the proof process if desired.

The functions `init-analysis` and the user functions `bc` and `bcs` all accept the following keywords:

`cutoffs` Cutoff test as described above.

`succeeds` Success test as described above.

`succeeds-with-bound` Also as described above.

`terminate` Termination test as described above.

`initial-value` The truth value assigned to the query in the absence of database information.

`analysis` If supplied, use this analysis instead of creating a new one.

8.2 Forward chaining

Forward chaining in MVL is not as simple as it is in many other logic programming languages because of the nonmonotonic nature of multivalued inference. It follows from this that inserting a new sentence into the database may cause a previous conclusion to be retracted, and this may in turn cause other sentences to become unsupported.

There are two functions that use this information to modify the database. The more common is called `fc` and accepts as an argument a proposition to be added to the database. The proposition is added and provided that it was not already in the database when the forward chainer was invoked, its consequences are also added to the database and forward chained upon. The `fc` function accepts the following keywords:

`value` This keyword is similar to that used by `stash` and `state`.

`increment-p` This keyword is also similar to that used by `stash` and `state`. The default value is `NIL`.

`fc-increment-p` When the information content of some sentence in the database has *fallen*, should this drop be propagated to other sentences that appear to depend on the sentence in question? The default value is `T`.

`cutoffs` Only forward chain if the sentence being inserted into the database has a truth value that passes the test corresponding to `cutoffs`. (See Section 12.2.2.) The default value used is the one associated with the active bilattice.

`include-modal` If `T` (the default), then the forward chainer also looks for rules that fire when the truth value of a modal expression involving p changes (where p is the sentence being forward chained upon). See Section 15. This keyword is included because searching for rules of this form is computationally expensive.

Slightly less common is `erase`, the function used to remove a proposition from the database. This function accepts a database sentence as an argument, and also the keywords `value`, `decrement-p` and `cutoffs`, which are similar to the corresponding keywords for `fc`.

Summarizing:

`fc(p)` Inserts the sentence p into the database and forward chains. Allowed keywords: `value`, `increment-p`, `fc-increment-p`, `cutoffs` and `include-modal`.

`erase(p)` Removes p from the database and forward chains. Allowed keywords: `value`, `decrement-p` and `cutoffs`.

8.3 Debugging tools

Debugging facilities are limited in MVL. The facilities that exist describe the prover's activities during backward-chaining, forward-chaining, or while the first-order theorem prover is looking for a conventional proof of some proposition.

8.3.1 Backward chaining

Diagnostics are displayed during backward chaining using the two global variables `*bc-trace*` and `*anytime--trace*`.

Whenever a task is encountered that involves a proposition that is an instance of some element of `*bc-trace*`, information concerning the task will be displayed to the user. The variable `*anytime-trace*` is similar, but only reports when an *inference* task is encountered. An inference task is one that is trying to prove something, as opposed to one analyzing a modal operator; this distinction is described in greater detail in the file bc.lisp. In practice, the distinction between `*bc-trace*` and `*anytime-trace*` is that the former should be used for debugging purposes, while the latter should be used if you are interested in watching the prover approximate and then refine its eventual answer to some declarative query.

Although the user can manipulate `*bc-trace*` and `*anytime-trace*` directly, there are four functions written to help do so:

`bc-trace(pat)` Include `pat` on `*bc-trace*`, so that any instance of `pat` will be traced.

`bc-untrace(pat)` Remove any instance of `pat` from `*bc-trace*`.

`anytime-trace(pat)` Include `pat` on `*anytime-trace*`, so that any instance of `pat` will be traced.

`anytime-untrace(pat)` Remove any instance of `pat` from `*anytime-trace*`.

For all of these functions, the argument is optional. If omitted, `(?*)` (of which everything is an instance) is used.

8.3.2 Forward chaining

Tracing the activity of the forward chainer is essentially identical to tracing the behavior of the backward chainer; the global variable `*fc-trace*` is used and is manipulated using the functions:

`fc-trace(pat)` Include `pat` on `*fc-trace*`, so that any instance of `pat` will be traced.

`fc-untrace(pat)` Remove any instance of `pat` from `*fc-trace*`.

Once again, the argument is optional.

8.3.3 The first-order prover

Tracing the activity of the first-order theorem prover itself is somewhat different. The facility for doing this is similar to the Common Lisp `trace` facility, but instead of reporting when a function is invoked, it reports when an attempt is made to prove an instance of a particular predicate, and also when such an instance is found successfully in the database.

The functions used to control this are `bc-watch` and `bc-unwatch`. They are similar to `trace` and `untrace`:

`bc-watch(preds)` Accepts as arguments any number of predicates that should be watched as the prover proceeds. Each predicate can also be a list of the form `(predicate depth)` where `depth` is the number of successive ancestors to be displayed when the prover encounters an instance of the predicate. (This can be used to understand *why* a particular clause is relevant to the proof being attempted.) `bc-watch` returns a list of the predicates currently being WATCHed.

`bc-unwatch(preds)` Removes the given predicates from the list of WATCHed predicates. If no arguments are supplied, all predicates are unWATCHed.

There are two additional features involved in tracing the activities of the first-order prover. The first is used whenever the global parameter `*save-nodes*` is `T` (which it is by default). When the first-order prover displays information regarding a particular proof node, the node is labelled with a number. Subsequent nodes will then use this label if they need to refer to the node in question (e.g., as a parent).

If `*save-nodes*` is `T`, then `*break-node*` can be set to an integer; when a node is displayed that is labelled with this integer, a `(break)` will be executed. This is to allow the user to interrupt the first-order prover if he needs to examine the proof stack or some other MVL internal.

`*save-nodes*` If `T` (the default), nodes are labelled with consecutive integers when they are displayed by the first-order prover.

`*break-node*` The number of a node at which the proof process should be interrupted.

8.4 The graphical interface (Allegro Common Lisp only)

On systems running Allegro Common Lisp, an interface has been provided that displays the current analysis using a tree representation. There are two types of viewers, the static viewer and the dynamic viewer. The static viewer is called when the proof is over to see how MVL achieved its results. The dynamic viewer, on the other hand, shows MVL in the process of performing the proof. This section describes how to use the viewers and the extra features they provide.

8.4.1 The static viewer

Invoking the static viewer The static viewer is invoked using the following function:

`static-display(a)` Invoke the static viewer, where a is the analysis structure returned by MVL when the proof is complete.

Calling `static-display` brings up two windows: the viewer and the overviewer. The viewer shows the proof tree, where the nodes are tasks in the proof. Since proofs can be large, only a portion of the proof is visible in the viewer at any time. The overviewer window contains a reduced picture of the whole proof tree. The section that is currently visible in the viewer is marked by a black rectangle.

`Static-display` closes any windows left from a previous viewer session. Thus, only one viewer can be active at any time.

Moving in the viewer The overviewer is used to move in the tree. To display a node, click the left button on it and it will be displayed by the viewer; the black rectangle moves to show the new area displayed. If the left button is held down while the mouse is moved in the overviewer, the visible area is dragged across the proof tree.

Scroll bars are also available for the viewer and can be activated by setting the variable `*viewer--scroll-bars*` to `T` before calling `static-display`.

Selecting tasks to display The viewer can display all tasks in the proof or just tasks of a certain type. The following functions should be called before `static-display` and tell the viewer what to display:

- `static-bc-trace`
- `static-bc-untrace`
- `static-anytime-trace`
- `static-anytime-untrace`

These functions are exact counterparts to the functions `bc-trace`, etc.

The viewer display The viewer shows tasks and connections between tasks. A task is connected to a task below it if it is its parent. A task is represented by an icon and some text as follows:

Line 1 The task's proposition, ending with ... if there is insufficient room to display it.

Line 2 The task's truth value, once again truncated if need be.

Icon This shows the type of the task, as follows:

- Inference task: a square. The square is empty if the task is active, and filled with an X otherwise.
- Modal-clause task: a filled circle.
- Modal task: a filled, inverted triangle.

An arrow appears to the left of the icon if the task has been selected by the user.

A number appears to the right of the icon if the task's proof is shared by at least one other task. The tasks that share the proof are labeled with the same number.

Viewer menus The viewer provides two menus, one performing general operations on the whole display (the "viewer menu") and one performing operations on tasks (the "task menu").

The viewer menu pops up when the right button is clicked while pointing at empty space in the window. It provides the following features:

1. **Display Root** moves the visible area to make the root of the tree visible. This feature can also be invoked by ⟨ctrl⟩-L (pressing the left button while holding the control key down).

2. **Show Overviewer** brings the overviewer window to the surface. Also invoked by ⟨shift⟩-L.

3. **Repaint** redraws the viewer and can also be invoked by pressing the left button.

4. **Flush** closes all windows related to the viewer.

When the mouse is clicked on a task the task menu is popped up. A rectangle frames the two text lines of a task when the mouse is over it, making it is easy to tell when the task menu will appear.

The task menu provides the following features:

1. **Inspect** brings up a composer window to inspect the task. Also invoked by by pressing the left button while the mouse is over a task.

2. **Root** moves the visible region so that the task is at its top center.

3. **Center** moves the visible region so that the task is at the middle of the screen.

4. **Display in Listener** prints the proposition and truth value of the task in the Listener. This is helpful when the whole text does not fit in the viewer. Also invoked by ⟨ctrl⟩-L when the mouse is over a task.

5. **Display in Separate Window** brings up the magnifier, a window that shows the full proposition and truth value of a task. Also invoked by ⟨shift⟩-L when the mouse is over a task.

6. **Setf `*C*` to Object** sets the variable `mvl::*C*` to the task at which the mouse is pointing.

The overviewer display The overviewer shows tasks as big points, connected by lines. Selected tasks are represented by slightly bigger points with a white dot in the middle.

Overviewer menus Like the viewer, the overviewer has menus of a general nature and menus dealing with single tasks.

The overviewer menu pops up when the right button is pressed on empty space (anything but a task). It provides the following features:

1. **Reorganize** calls `static-display` again to rearrange the nodes of the tree. The system does not respond while the tree is reorganized and the mouse icon changes to a watch to signal this.

2. **Repaint** clears the window and redraws it.

3. **Break** calls the Lisp break function as soon as possible during dynamic viewing.

The overviewer task menu appears when the right button is pressed and the mouse is over a task in the tree. If the tree is dense, it is a bit hard to tell which task has been selected. For this reason, the menu pops up with its top left corner on the selected task.

The overviewer task menu provides the following features:

1. **Select** selects the task.

2. **Deselect** deselects the task.

3. **Deselect All** deselects all selected tasks.

4. **Setf `*C*` to Object** sets the variable `mvl::*C*` to the task.

Selection and deselection can also be done by pressing the middle mouse button on a task in the overviewer. This button toggles selection on and off.

Task selection As described above, the viewer allows the selection of tasks. A selected task is distinguished graphically from an unselected one in both the viewer and the overviewer.

The following functions are also provided to select and unselect tasks:

`select-tasks(tasks)` Select the given tasks.

`deselect-tasks(tasks)` Deselect the given tasks.

`selected-tasks()` Returns a list of all selected tasks.

8.4.2 The dynamic viewer

The dynamic viewer shows MVL's progress as it proves a proposition. Again the viewer and the overviewer are shown, but the tree in the overviewer grows as the proof progresses. During dynamic viewing an empty rectangle is used to show the region of the overviewer that is visible in the viewer. When the proof is completed, dynamic viewing ends and this rectangle becomes a filled rectangle.

The dynamic viewer also shows the propagation of truth values up the tree. Each time a task changes value, it blinks rapidly in the overviewer.

Invoking the dynamic viewer To invoke the dynamic viewer, the following function must be called before `bc` or an analogous function is invoked:

`dynamic-view()` Instructs MVL to use the dynamic viewer to show the progress of the prover.

To make MVL run without the dynamic viewer, call the following function:

`dynamic-unview()` Instructs MVL to cease using the dynamic viewer when the prover is invoked.

Selecting tasks to display The dynamic viewer can display all tasks in the proof or just tasks of a certain type. The following functions should be called before invoking the prover and tell the viewer what to display:

- `dynamic-bc-trace`
- `dynamic-bc-untrace`
- `dynamic-anytime-trace`
- `dynamic-anytime-untrace`

These functions are exact counterparts to the functions `bc-trace`, etc.

Behavior of the dynamic viewer All features in the section on the static viewer can be used during dynamic viewing. In this section we discuss only features and observations relevant to the dynamic viewer alone.

The dynamic viewer adds small subtrees trees to the growing proof tree. During these additions, the data structures used may be corrupted and so user actions are not processed until the subtree addition is completed. There may therefore be a small delay before a user request is processed.

It is important to note that interrupting the dynamic viewer with a double ⟨ctrl⟩-C is not a good idea because the graphic system often cannot be restarted. This is the reason for the appearance of the **Break** item in the overviewer menu.

The dynamic viewer tries to maintain the tree in its tidiest possible form, often repainting portions of the tree to do so. If, in spite of this, the tree gets too messy to be useful, the menu item **Reorganize** can be used to rearrange the tree more sensibly.

During dynamic viewing, if the truth value of a task is displayed in a separate window, this truth value will be updated whenever it changes. This "magnifier" window can be flushed using the magnifier menu which pops up when the right button is pressed on the magnifier window.

8.4.3 Customizing the viewer

The following variables can be set by the user to alter the behavior of the viewer:

`*viewer-size*` A list of the form (width height), this specifies the initial size of the viewer window.

`*overviewer-size*` The same for the overviewer window.

`*viewer-scroll-bars*` When set to `T`, the viewer's scroll bars become active and can be used. The default value is `NIL` since the overviewer provides a better way to move around the tree. This variable has to be set before the viewer window is created in order to have any effect.

9 Procedural attachment

In some cases, it may be desirable to store a fact in the database *without* using the functions supplied in MVL. Suppose, for example, that we have an $n+1$-ary relation R that corresponds to an n-ary function f, so that $R(x_1,\ldots,x_n,y)$ holds just in case $f(x_1,\ldots,x_n)=y$. Now, if we assert $R(c_1,\ldots,c_n,d)$ into the database, where d and the c_i are constants, we should probably also *remove* any sentences of the form $R(c_1,\ldots,c_n,d')$, where $d'\neq d$. This sort of a situation might arise if we were monitoring the temperature in a room, and represented the observed temperature using a database sentence such as

`(TEMPERATURE ROOM 72)` (3)

We handle this by informing MVL that to stash a sentence of the form `(TEMPERATURE ...)`, it should use not the usual function `stash`, but some other user-defined function, say `unique-stash`. This new function would first `unstash` any sentence that agreed with the given one except for the value of the final term, and would then invoke `stash` on the original sentence, but with `allow-procedural-attachment` set to `NIL` so that `unique-stash` was not invoked a second time.

To modify the function used by MVL to stash sentences such as this, the user should simply place the desired function (in this case, `unique-stash`) on the `stash` property of the relation in question (in this case, `TEMPERATURE`).[3] The system will then automatically invoke `unique-stash` instead of `stash` when a fact such as (3) is asserted.

The functions within MVL that can be modified in this fashion are `stash`, `unstash`, `lookup` and `lookups`. In addition, the procedural attachments for `lookup` and `lookups` are linked, so that if `lookup` has a procedural

[3] As described in Section 11.3, the stash property of a relation can also be an association list, if multiple invocations of the system are running in parallel.

attachment and `lookups` does not, a call to `lookups` will invoke the procedural attachment for `lookup` and then pluralize the result. Similar remarks hold if `lookups` and not `lookup` has a procedural attachment.

The procedural attachment is effected using the following functions:

`invoke(fn p)` This function invokes `fn` on the sentence *p*. In addition, any keyword arguments are passed to `fn`, but with `&allow-other-keys` so that the keywords will be ignored if `fn` is not expecting them.

`lookup-call(fn p)` This function is similar to `invoke`, but modifies the values returned to make them agree with those expected to be returned by `lookup`. Unless the function `fn` supplies them, the truth value is assumed to be `true`. The second value is `attach`.

`lookups-call(fn p)` The plural version of `lookup-call`.

9.1 Facilities to create procedural attachments

Some facilities exist to create procedural attachments within the MVL system. The following two macros are used for this purpose:

`lisp-predicate(pred)` This macro is used to indicate that the given predicate can be evaluated by a LISP call. It will also cause MVL to signal an error if an attempt is made to stash or unstash an instance of the function.

`lisp-defined(fn)` This macro is used to indicate that the given function can be evaluated by a LISP call. It will also cause MVL to signal an error if an attempt is made to stash or unstash an instance of the function. `Lisp-defined` also accepts an optional argument, which is an inverter function. The overall result of attempting to lookup an instance of the function can be described as follows:

1. If none of the arguments to the function (i.e., all but the last of the arguments of the relation) is a variable, then the function is called and the result is unified with the final argument of the relation.
2. If the inverter function has been supplied and is equal to `T`, then the function is called and the result is unified with the final argument of the relation even if some of the arguments are variables.
3. If some of the arguments are variables but the result is not a variable, then the inverter function is invoked if one has been supplied. The arguments passed to the inverter function are the result (i.e., the last argument of the relation), and a list of all of the arguments (i.e., all but the last argument of the relation). If there is no inverter function, then `NIL` is returned. In addition, if `*warn-on-bad-lisp-invocations*` is not `NIL`, the user is warned. (The parameter `*warn-on--bad-lisp-invocations*` defaults to `NIL`.)
4. If both some of the arguments and the result are variables, then `NIL` is returned. In addition, if `*warn-on-bad-lisp-invocations*` is not `NIL`, the user is warned.

The current version of MVL invokes `lisp-predicate` on the following LISP predicates: `numberp`, $<$, $>$, `equal`, `null`, `atom`, `listp`, `varp`, `varp?`, `varp*`, `groundp`, and $=$. The procedural attachment to $=$ attempts to unify its arguments.

The macro `lisp-defined` is invoked on the following: `car`, `cdr`, `cadr`, `cons`, `list`, `list*`, `revappend`, `append`, `reverse`, `length`, `assoc`, `delete`, `+`, `*`, `-`, `/`, `floor`, `ceiling`, `truncate`, `round`, `float`, `mod`, `rem`, `max`, `min`, `sin`, `cos`, `atan`, `sqrt`, and `eval`.

9.2 Other procedurally attached relations

In addition to the LISP functions and predicates described in the previous section, the following relations have been defined and are procedurally attached to suitable LISP functions:

`(bagof ?x ?p ?b)` This relation holds for x, p and b if and only if b is a list of all x satisfying p.

`(done exp-1 ... exp-n)` This predicate, which is useful for stashing only, causes LISP to evaluate each of the subsequent expressions.

`(value symbol val)` Stashing this predicate sets the given symbol to the value `val`. Unstashing it makes the symbol unbound if its current value unifies with `val`.

`(property symbol value indicator)` Stashing this predicate sets the symbol's `indicator` property to `value` for the current invocation of MVL. Unstashing it removes the `indicator` property for the current invocation of MVL if the previous value of this property unifies with `value`. (See the definitions of `get-mvl` and `rem-mvl` in Section 11.3.)

`(known p)` Stashing this predicate simply stashes p; unstashing it causes an error. Looking it up looks up p.

`(unknown p)` Stashing this predicate unstashes p; unstashing it causes an error. Looking it up fails if p can be found in the database, and succeeds otherwise.

`(member item list)` Tries to unify `item` with the various elements in `list`.

10 Testing the MVL system

10.1 As a diagnostic tool

There are two functions available to help you to test both MVL and any applications you write using it. The first, `test-mvl`, is one you should already have run; if MVL has been installed successfully, it should return 0.

`Test-mvl` works by invoking the generic testing function `mvl-test-file` on the file `mvl.test` in the MVL directory.[4] This file also contains documentation regarding the details of the various tests being executed.

In general, `mvl-test-file` accepts as input a file name, and looks for a file with an extension `.TEST` on either the MVL or the current directory. It then works through the file by evaluating the first s-expression it contains, and comparing the value returned to the second s-expression. If these two s-expressions match, the next s-expression is evaluated, and so on. If there is a mismatch, `mvl-test-file` reports the error; eventually, it returns the total number of errors found in the entire file. When doing the comparison, `**` acts as a wildcard. Thus, if the second s-expression is simply `**`, it will always match; if it is `(** ** **)`, it will match whenever the first expression returns a list of three values, and so on.

`test-mvl` Tests the MVL system by invoking the function `mvl-test-file` on the file `MVL.TEST`.

`mvl-test-file(string)` Can be used to evaluate various forms by the MVL system and compare the results to predefined values. Both the forms and the expected answers are in the `.TEST` file whose name is supplied to the function. Returns the number of errors found.

10.2 The MVL tutorial/demo

MVL is also equipped with a tutorial to help you get to know the system; in can be invoked with the function `demo-mvl`. This tutorial works you through a variety of examples showing the flexibility of multivalued inference and its application to a variety of examples.

As with `test-mvl`, the demo program works by invoking a function on a demo file (as opposed to a test file); the function is `mvl-demo-file` and expects a file name as argument; the file is expected to end in

[4] `Test-mvl` works in its own invocation of MVL, so that the existing database is not modified. This is not true of `mvl-test-file`, however.

.DEMO. The demo file is expected to contain a variety of s-expressions, and the demo proceeds simply by displaying these expressions and then showing what they return as values. The demo file can also contain the following special forms:

1. **quiet**. Read and evaluated the next form (presumably for side effect), but do not display the form or the result to the user.
2. **pause**. Pause and prompt the user for a carriage return to continue. The allowed responses to the interruption are:
   (a) A carriage return continues with the demo.
   (b) stop exits the demo.
   (c) Any other input is displayed and evaluated as if it were included in the demo itself.

If you run the MVL demo, you will see that it begins as follows:

```
Demonstrating backward chainer.

Evaluating: (LOAD-LOGIC f)
Returns:    First-order logic.

Evaluating: (MVL-LOAD f1)
Returns:    DONE

Enter <CR> to continue:
```

The program informs you that it is beginning by demonstrating the backward chainer, and then loads the bilattice corresponding to first-order logic. Then it loads the facts in the file **f1.mvl**; it doesn't display these facts, but you can find them by examining that file directly. The demo program now pauses. If you type **(contents)**, that function will be invoked and you'll see what the MVL database currently contains; if you type **stop**, you'll exit the demo. Entering a carriage return causes the tutorial to continue. The contents of the database at this point can also be found directly in the file **f1.mvl** and are described in Section 15 and Appendix B.2 of the paper "Multivalued logics: A uniform approach to inference in artificial intelligence" as well. The database itself gives a declarative description of a full adder.

Continuing the demo, we next evaluate

```
(get-bdg '?x (answer-binding (bc '(val (out 1 f1) 1 ?x))))
```

This invokes the backward chainer on the sentence

```
(val (out 1 f1) 1 ?x))
```

to see if we can prove that the first output of the full adder takes some value ?x. Then we get the binding of this variable to see what the value actually is – OFF, as it turns out. The remainder of the demo is similar; many of the examples in these files can also be found in the various technical papers describing the system.

demo-mvl Demonstrate the MVL system by invoking the function **mvl-demo-file** on the file MVL.DEMO.

mvl-demo-file(string) Can be used to construct new demos or tutorials.

11 System support

11.1 Functions that compile and load the MVL system

The following three functions can be used to manipulate the MVL system as a whole:

`load-mvl` Loads the MVL system.

`compile-mvl` Compiles and loads the MVL system. On an Allegro system, the keyword `recompile` forces a recompile of all of the files and the keyword `reload` is ignored (since the files will all be reloaded anyway). On a Symbolics system, the keyword `reload` forces all of the files to be reloaded and the keyword `recompile` forces them all to be recompiled.

`release-mvl` (Symbolics only) Compiles and loads the MVL system, and declares the new version to be `released`. The keyword `no-recompile` can be used to skip the compilation step.

11.2 The MVL package

In order to avoid name conflicts with existing code, the MVL system resides in its own package, `MVL`. All of the symbols and functions appearing in the appendix are exported from this package and can therefore be accessed by invoking the function `use-package(mvl)`. When a new bilattice is loaded, the modal operators it defines are also exported.

To use the MVL system, invoke the function `(use-package 'mvl)` to make these symbols accessible.[5]

11.3 Running several copies of MVL in parallel

Some users may wish to run several copies of the MVL system simultaneously, perhaps associating them with distinct LISP processes or with individual agents. (The use of multiple copies of MVL to represent distinct components of a single agent's knowledge goes against the grain of the multivalued approach and is not recommended.) In order to support this facility, the following functions and macros have been defined:

`existing-mvl-invocation` This function returns a single value that is an instance of the structure `mvl-invocation`. It is useful primarily for saving the current copy of MVL so that it can be restored later.

`create-mvl-invocation` This function sets up a new copy of MVL. It accepts an optional argument that is an instance of `mvl-invocation`; this can be used to reset the state of MVL to a previous one. If the optional argument is omitted, default values are used. The value returned is the MVL invocation that was active before the function was called, so that another call to `create-mvl-invocation` can be used to restore the original environment.

`with-mvl-invocation` This is a macro similar to the Common Lisp macro `with-open-file`. It accepts an optional argument that is an instance of `mvl-invocation` and is followed by a body of forms (presumably including a `load-logic`) that are evaluated in a locally-bound copy of MVL.

These functions work by locally or globally rebinding the values of the various parameters used by the MVL system (`*active-bilattice*`, `true` and so on). The only difficulty with this is that procedural attachments for a relation R use the property list of R, and this cannot be rebound.

In order to cater to this difficulty, there is an additional global variable `*mvl-invocation*` that labels the currently active copy of MVL. To determine what procedure is attached to the `stash` operation for R, the `stash` property of R is examined. There are now the following possibilities:

1. If the value is `NIL`, `stash` is not procedurally attached, so the usual MVL function is called.

[5] As an alternative, `(in-package "mvl")` will put you in the MVL package itself.

2. If the value is an atom, then it is assumed that this atom is the name of a function that is the procedural attachment for `stash` independent of the copy of MVL that is currently active.

3. If the value is a list, it is treated as an association list and a cons whose car is `*mvl-invocation*` is searched for.

   (a) If such a cons is found, its cdr is used as the procedural attachment to `stash`.

   (b) If no such cons is found, but there is a cons whose car is `T`, then the cdr of that cons is used.

   (c) If no cons has either `*mvl-invocation*` or `T` as its car, then `stash` is assumed to be unattached.

The other functions described in Section 9 are handled similarly.

The association lists described above are manipulated using the following functions:

`get-mvl (symbol indicator)` Like the Common Lisp `get`, but if the value is a list, treats it as an a-list and searches for an item whose car matches `*mvl-invocation*`. The value taken can be changed with `setf`.

`rem-mvl (symbol indicator)` Like the Common Lisp `remprop`, but if the value is a list, treats it as an a-list and removes the item whose car matches `*mvl-invocation*`.

12 Selecting a bilattice

12.1 Bilattice structure

Of course, most of the power of the MVL system comes not from the features that were described in Sections 3–11 of this manual, but from the fact that the user can change representation language conveniently within the system. He does this by changing the *bilattice* that he is using.

A bilattice is the mathematical structure from which the truth values used by the system are selected, and the formal ideas underlying this approach are discussed at length in the accompanying technical papers. In the MVL system, a bilattice is described as an instance of the `bilattice` structure. This structure has 26 fields; of these, 16 are used to provide the formal information required by the system, and the others are used to simplify the user interface. Here are 13 of the fields used to describe the bilattice itself:

`true` A LISP object that is the truth value assigned to a sentence that is unconditionally true, such as the tautology `(OR (P) (NOT (P)))`.

`false` A LISP object that is the truth value assigned to a sentence that is unconditionally false, such as the negation of a tautology.

`unknown` A LISP object that is the truth value assigned to a sentence about which nothing is known.

`bottom` A LISP object that is the truth value assigned to a sentence that is both true and false. If any sentence is assigned this truth value (or can be shown to have it), it indicates the presence of a contradiction in the database.

`eq` A LISP function that takes two truth values as arguments and returns `T` if they are equal and `NIL` if they are not. This function is needed because it may be possible to represent a single element of the bilattice as a LISP object in a variety of fashions.

`and` A LISP function that takes two truth values as arguments and returns the truth value corresponding to their conjunction. If the two sentences p and q are labelled with truth values v_1 and v_2 respectively, then the conjunction of the truth values is the truth value that can be assigned to $p \wedge q$.

`or` A LISP function that takes two truth values as arguments and returns the truth value corresponding to their disjunction.

`not` A LISP function that takes a truth value as its argument and returns the truth value corresponding to its negation.

`plus` A LISP function that takes two truth values as arguments and returns the truth value corresponding to their "sum." If there is reason to believe that a sentence p should be labelled with truth value v_1, and independent reason to believe that it should be labelled with truth value v_2, then the actual label is given by the sum $v_1 + v_2$. This operation therefore corresponds to combination of evidence.

`dot` A LISP function that is the dual of the `plus` operation. It takes two truth values v_1 and v_2 as arguments and returns the greatest truth value that could subsequently be strengthened to a conclusion of either v_1 or v_2. This is a "consensus" operation.

`kle` A LISP function that takes two truth values x and y and determines whether $x + y = y$. Although this can also be done using the bilattice $+$ and $=$ operations, this field can be provided to give a more efficient implementation of this operation. This field is optional.

`tle` A LISP function that takes two truth values x and y and determines whether $x \vee y = y$. This field is also optional.

`modal-ops` A list of the modal operators associated with the given bilattice. See Section 15.

Three of the remaining formal slots are concerned with the incorporation of binding information into truth values. In some cases, when a proposition is retrieved from the database, it is only an instance of the database proposition that is used, and the truth value may have to be modified to reflect that:

`plug` This LISP function accepts as arguments a truth value and a binding list, and returns a potentially modified truth value. The default value is `NIL` and is interpreted as a function that leaves the truth value unchanged.

There is also a function that accepts a truth value and returns the variables appearing within it:

`vars` This LISP function accepts a truth value as its argument and returns a list of the variables appearing in it. The default value is `NIL` and is interpreted as a function that returns `NIL` independent of its argument.

The function `absorb` is needed for subtle technical reasons when working with bilattices with non-`NIL` plugging functions; further information on this can be found in the file `load.lisp`. All three of these functions are `NIL` if there is no interaction between the bilattice truth values and the variables that they contain.

The remaining ten fields in the `bilattice` structure are as follows:

`dws` This LISP function, "dot with star," accepts as arguments two truth values, v_1 and v_2. The truth value returned is v_1, modified so as to remove any contribution that could be attributed to a sentence with truth value v_2.

`stash-val` A function that, given a sentence p, returns the default truth value to be passed to `stash`.

`unstash-val` A function that, given a sentence p, returns the default truth value to be passed to `unstash`.

`cutoff-val` The default cutoff point that must be exceeded by any sentence if it is to be returned by `lookup`. See Section 12.2.2.

`print-fn` This function is used to pretty-print truth values in the associated bilattice. Its three arguments are a truth value, the bilattice in question, and a stream.

`long-desc` A long description of the bilattice, used for input and output only.

`short-desc` A short description of the bilattice, also used for input and output only.

`char` A one-character identifier for the bilattice. This character can be used to change the bilattice using the function `load-logic` described in Section 12.3.

`type` An overall description of the mathematical form of the bilattice. Perhaps it is a product of two other bilattices, or formed from them in some other fashion. This defaults to `basic` if no other value is supplied; various methods for constructing new bilattices from old ones are described in Section 14.

`components` A list of the components used to construct a bilattice whose `type` is not `basic`.

12.2 The active bilattice

At any particular point in time, there will be a single bilattice that has been selected by the user to be used in the application he is developing. This bilattice is stored as the value of `*active-bilattice*`; in addition, many of the slots of the active bilattice are directly available to the user. Here they are:

`*active-bilattice*` The currently active bilattice.

`true` The truth value `true` in the active bilattice.

`false` The truth value `false` in the active bilattice.

`unknown` The truth value `unknown` in the active bilattice.

`bottom` The truth value `bottom` in the active bilattice.

`std-cutoffs` Cutoff information constructed from the value of `cutoff-val` in the active bilattice. See Section 12.2.2.

`stash-value` The function `stash-val` in the active bilattice.

`unstash-value` The function `unstash-val` in the active bilattice.

`mvl-plus` The function `plus` in the active bilattice.

`mvl-dot` The function `dot` in the active bilattice.

`mvl-and` The function `and` in the active bilattice.

`mvl-or` The function `or` in the active bilattice.

`mvl-not` The function `not` in the active bilattice.

`mvl-eq` The function `eq` in the active bilattice.

`dot-with-star` The function `dws` in the active bilattice.

`mvl-vars` The function `vars` in the active bilattice.

`mvl-plug` The function `plug` in the active bilattice.

`mvl-print` This function accepts a single argument, a truth value, and modifies it in a way that will cause it to be pretty-printed. The value returned by mvl-print should not be used for any purpose other than printing.

All of the functions above accept an optional argument that can be used to evaluate their result using some bilattice other than the active one.

12.2.1 Functions derived from the active bilattice

There are a variety of functions associated with the functions that are defined when a new bilattice is made active. In order to simplify the description of these functions, we will denote `mvl-and` by $\wedge$, `mvl-or` by $\vee$, `mvl-not` by $\neg$, `mvl-plus` by $+$, and `mvl-dot` by $\cdot$. The first auxiliary function defined is `dot-with-not`:

`dot-with-not(x)` Computes $x \cdot \neg x$.

The other functions are constructed from the partial orders defined using the bilattice operations. We will write $x \leq_t y$ just in case $x \wedge y = x$, for example, and $x \leq_k y$ just in case $x \cdot y = x$. (All of this notation follows that in the paper, "Multivalued logics: A uniform approach to inference in artificial intelligence".)

Given these definitions, the following LISP functions are defined:

`k-le(x y)` Returns `T` if $x \leq_k y$, and `NIL` otherwise.

`k-ge(x y)` Returns `T` if $x \geq_k y$, and `NIL` otherwise.

`t-le(x y)` Returns `T` if $x \leq_t y$, and `NIL` otherwise.

`t-ge(x y)` Returns `T` if $x \geq_t y$, and `NIL` otherwise.

`k-lt(x y)` Returns `T` if $x <_k y$, and `NIL` otherwise.

`k-gt(x y)` Returns `T` if $x >_k y$, and `NIL` otherwise.

`t-lt(x y)` Returns `T` if $x <_t y$, and `NIL` otherwise.

`t-gt(x y)` Returns `T` if $x >_t y$, and `NIL` otherwise.

`k-not-le(x y)` Returns `T` if $x \not\leq_k y$, and `NIL` otherwise.

`k-not-ge(x y)` Returns `T` if $x \not\geq_k y$, and `NIL` otherwise.

`t-not-le(x y)` Returns `T` if $x \not\leq_t y$, and `NIL` otherwise.

`t-not-ge(x y)` Returns `T` if $x \not\geq_t y$, and `NIL` otherwise.

`k-not-lt(x y)` Returns `T` if $x \not<_k y$, and `NIL` otherwise.

`k-not-gt(x y)` Returns `T` if $x \not>_k y$, and `NIL` otherwise.

`t-not-lt(x y)` Returns `T` if $x \not<_t y$, and `NIL` otherwise.

`t-not-gt(x y)` Returns `T` if $x \not>_t y$, and `NIL` otherwise.

All of these functions accept an optional argument that is the bilattice to be used in the comparison. This argument defaults to the currently active bilattice.

12.2.2 Cutoff information

We will refer to each of the 16 functions defined at the end of the previous section (e.g., excluding `dot-with-not`) as a *tag*. Thus `k-le` is a tag, and so on.

We will now define a *simple test* to be a dotted pair

```
(v . tag)
```

where `v` is a truth value and `tag` is a tag. We will say that a truth value `x` *satisfies* the simple test if and only if the function `tag`, when applied to the arguments x and v (in that order), returns `T`.

As an example, consider the simple test

```
(true .  t-ge)
```

This test will be satisfied by any truth value that is $\geq_t$ **true**. Since **true** itself is the only truth value satisfying this test, the test will be satisfied for **true** and no other truth values. This is the test that is frequently used to decide to terminate a proof effort.[6]

Alternatively, the test used to determine whether or not to return an answer after the theorem prover runs to completion is often given by

```
(unknown .  k-gt)
```

This test will succeed for any truth value about which more is known that `unknown` – in other words, for any truth value other than `unknown` itself.

In general, the simple tests are combined into *cutoffs.* A cutoff is simply a list of lists of simple tests, and is interpreted as being in conjunctive normal form. Thus the cutoff corresponding to the simple test s is given simply by

```
((s))
```

A cutoff corresponding to the conjunction of tests $s_1, \ldots, s_n$ is given by

```
((s-1) ...  (s-n))
```

while that corresponding to their *disjunction* is given by

```
((s-1 ...  s-n))
```

In general, we can build up conjunctions of disjunctions of simple tests in as complex a manner as we wish. The cutoff `NIL` corresponds to the empty conjunction and therefore always succeeds; the cutoff `(NIL)` contains an empty disjunction and therefore always fails. For convenience, these two cutoffs are stored in the global constants **`*always-succeeds*`** and **`*never-succeeds*`** respectively.

Given a truth value x and a cutoff c, the expression **`(test x c)`** evaluates to `T` if x passes the test, and to `NIL` if it fails it. There is also a function that negates cutoffs:

`test(value cutoffs &optional bilattice)` Returns `T` if the truth value passes the test associated to **`cutoffs`**, and `NIL` otherwise. The test is performed for the supplied bilattice, or for the currently active one if the optional argument is omitted.

`cutoffs-not(cutoffs &optional bilattice)` Returns a new cutoffs c such that

`(test x c)` = **`(test (mvl-not x) cutoffs)`**

for all truth values x. In other words, x passes c if and only if $\neg x$ passed the cutoffs supplied originally.

12.3 Selecting a new bilattice

To change the bilattice being used, simply invoke the function **`load-logic`**. There are three ways to specify the new bilattice:

1. Do nothing. You will then be asked to select the bilattice from the list of bilattices known to the system.

2. Supply the one-character identifier of the new bilattice as an argument to **`load-logic`**. The new bilattice must already be known to the system.

[6] Tests in the MVL system are in fact LISP structures with value and tag fields; we are using dotted lists in this manual only to make the presentation clearer.

3. Supply the new bilattice itself as an argument to `load-logic`.

`Load-logic` empties the database before the bilattice is changed.

The bilattices known to the system are kept in the global variable `*bilattices*`. To add a new bilattice to `*bilattices*`, invoke the function `bilattice` with the bilattice as an argument. This function will warn you if you are overwriting another bilattice that has the same one-character designator.

Summarizing:

`*bilattices*` A list of the bilattices known to the system.

`bilattice(b)` Add the bilattice b to `*bilattices*`.

`load-logic` Empty the database and make a new bilattice active. Accepts as an optional argument either a character designating a bilattice, or the new bilattice itself. If no argument is supplied, the user is prompted for the new bilattice.

13 Bilattices available in the MVL system

Here are the bilattices available in the MVL system as distributed. Each is discussed more fully in a subsequent section:

`*first-order-bilattice*` The bilattice corresponding to first-order logic.

`*atms-bilattice*` The bilattice corresponding to an ATMS, where the justification information is maintained in disjunctive normal form as suggested by de Kleer.

`*cnf-atms-bilattice*` An alternative version of the ATMS bilattice where the justification information is maintained in conjunctive normal form.

`*default-bilattice*` A bilattice that can be used for simple default reasoning.

`*hierarchy-bilattice*` A bilattice similar to the default bilattice but where the defaults are labelled with integer-valued strengths.

`*first-order-atms-bilattice*` This bilattice corresponds to a fully first-order ATMS, as opposed to the more conventional one dealing only with propositional logic.

`*circumscription-bilattice*` The bilattice used to compute the results of the circumscription axiom.

`*probability-bilattice*` An extended version of the ATMS bilattice that maintains probabilistic information as well.

`*time-bilattice*` A compound bilattice that can be used for reasoning about events occurring against a temporal background.

`*plan-bilattice*` A compound bilattice that can be used to solve planning problems.

13.1 First-order logic

The simplest bilattice is that corresponding to first-order logic, and contains only the four truth values `true` (displayed as `T`), `false` (displayed as `F`), `unknown` (`U`) and `bottom` (`T/F`). This bilattice is identified using the character **f**, and is active when MVL is first loaded.

13.2 Assumption-based truth maintenance

Except for the nonmonotonic and probabilistic ATMS bilattices, all of the bilattices corresponding to ATMS's have truth values that are dotted pairs `(j . k)`, where j is a justification for the sentence being labelled, and k is a justification for its negation. The bilattices differ in that the justifications are represented differently.

13.2.1 The ATMS bilattice

For the simplest ATMS bilattice, the justifications are simply disjunctive normal form expressions. Thus the justification

`((P23 P37) (P1))` (4)

is interpreted to mean that the sentence in question is a consequence either of `P1`, or of `P23` and `P37` in conjunction.

This bilattice is identified by the character **a**.

13.2.2 ATMS's in conjunctive normal form

Alternatively, we can represent the justifications in conjunctive normal form, as opposed to disjunctive normal form. This representation is useful, for example, in diagnosing multiple faults, since we want to be able to explain an observed failure *either* in terms of the collective failures of one set of components, or the collective failures of another set, and so on.

This bilattice is identified by the character **b**.

13.2.3 First-order ATMS's

It is also possible to extend these ideas slightly to obtain a fully first-order ATMS. In order to do this, we need simply modify justifications such as that appearing in (4) so that the primitive elements of the justification are not atoms referring to propositions, but pairs

`(atom . binding-list)`

where the atom refers to a proposition and the binding list indicates precisely what instantiation of that proposition is actually needed to justify the sentence in question.

As an example, suppose that an assumption in our database is `P1`, given by

`(FRIENDLY ?X)` (5)

and indicating that everything is friendly. If we also have the rule of inference

`(<= (HAPPY ?X) (FRIENDLY ?X))`

indicating that anything friendly is happy, and this rule is unconditionally true (i.e., not an assumption), then the truth value assigned to the query `(HAPPY HARRY)` will be

`((((P1 . ((?X . HARRY)))))`

indicating that Harry is happy by virtue of (5) applied to x = Harry, and that there is no justification for the conclusion that Harry is not happy. (Recall that all of the ATMS bilattices combine justification for and against each sentence into a single truth value.)

This bilattice is identified by the character **o**.

13.2.4 Probabilistic ATMS's

Truth values in the probabilistic ATMS bilattice consist of dotted pairs, where the `car` of a truth value is simply an element of the ATMS bilattice and the `cdr` is the probability that the given assumption is valid.

When entering an assumption, the user will be queried for the probability in question; the probabilities of compound sentences are then computed automatically by the system. The algorithm used is that described in Grnarov, Kleinrock and Gerla in the 1979 Proceedings of the Computer Networking Symposium.

The probabilistic ATMS bilattice is identified by the character **p**.

13.3 Default reasoning

The default bilattice consists of seven truth values. In addition to the usual elements `true`, `false`, `unknown` and `bottom`, there are elements `dt`, `df` and `*`.

The truth value `dt` labels a conclusion that is true by default (such as Tweety flying in the canonical example), while `df` labels a conclusion that is *false* by default. The final value, `*`, labels conclusions that are both true *and* false by default, as in the well-known Nixon-Republican-Quaker example.

This bilattice is identified by the character **d**.

13.3.1 Prioritized default reasoning

Another version of the default bilattice allows for priorities among the defaults. In this bilattice, truth values are of the form

```
(n . x)
```

where n is a nonnegative integer and x is an element of the first-order bilattice. Truth values with $n = 0$ are displayed as for the first-order bilattice; with $n = 1$ as for the default bilattice. For $n > 1$, we might see `D(3)T` or something like it. This particular truth value is the same as

```
(3 . true)
```

Truth values with small n override those with larger n. `Unknown` in this bilattice is given by the truth value `NIL`. The one-character identifier for the hierarchical default bilattice is **h**.

13.4 Circumscription

The default bilattices are not able to deal with all of the situations in which multiple extensions arise because single truth values such as `*` label sentences that are true in some extensions and false in others without providing any information about precisely which extensions are which. In order to handle this difficulty, we need to work with a more complex bilattice.

One way to view the default bilattice is as a copy of the first-order bilattice in which the point `unknown` is interpreted as, "not known with certainty," and is in fact replaced with another copy of the first-order bilattice that is used to record uncertain information (i.e., default truth, default falsity, or the presence of a default contradiction).

The circumscriptive bilattice is formed similarly, except that instead of simply recording default truth, falsity, and so on, we record a *justification* for default conclusions. This is done by replacing the point `unknown` of the first-order bilattice not with another copy of the first-order bilattice, but with a copy of the first-order ATMS bilattice. (This idea is in fact a general way to construct new bilattices from old ones, and is called a *surgery*. It is discussed in Section 14.3.) The designator for the circumscriptive bilattice is **c**. `Circum` and `circums` use this bilattice to accept a sentence as an argument and attempt to prove it from the circumscription axiom; they resemble `bc` and `bcs`.

The truth values in the circumscription bilattice corresponding to certain information are of the form `(T . x)`, where x is an element of the first-order bilattice. Those corresponding to uncertain information are of the form `(NIL . y)`, where y is an element of the first-order ATMS bilattice.

Finally, the circumscriptive theorem prover will output diagnostics concerning its progress if the global variable `*circum-trace*` has a non-`NIL` value.

Here is a summary of the circumscriptive facilities:

`*circum-trace*` If not `NIL`, the circumscriptive proof functions will output diagnostics describing their progress.

`circum(p)` Like `bc`, but looks for instances of p that follow from the circumscription axiom. Allowed keywords: `cutoffs` and `succeeds`.

`circums(p)` Like `bcs`, but looks for instances of p that follow from the circumscription axiom. Allowed keywords: `cutoffs` and `succeeds`.

13.5 Temporal reasoning

`*Time-bilattice*` is a bilattice of the truth values that might be assigned to propositions that have a time-dependent nature, such as whether or not a particular switch is on. The truth values represent the entire histories of the propositions, so that a single truth value is a complete function from a set of time points (taken to be the integers) into a set of "base" truth values (take to be elements of the nonmonotonic ATMS bilattice). This bilattice is discussed at greater length in the paper, "Computational considerations in reasoning about action," and a sample use of this bilattice can be found by examining the file `temporal.test` that is included with MVL.

13.6 Planning

Finally, `*plan-bilattice*` is a bilattice of the truth values that might be used in planning. The values are quite like those of `*time-bilattice*`, except they are functions from a set of plans (as opposed to the set of integeral times) into the first-order bilattice. This bilattice is discussed at greater length in the paper, "Computational considerations in reasoning about action II: Planning," and a sample use of this bilattice can be found by examining the file `plan.test` that is included with MVL.

14 Facilities to create new bilattices

There are a variety of ways in which it is possible to construct new bilattices from old ones, or from old ones in conjunction with instances of some other structure. In this section, we describe the extent to which these facilities exist within the MVL system.

14.1 Products

Perhaps the simplest way in which a new bilattice can be constructed is as the product of two existing bilattices. If A and B are bilattices, then a truth value in the product bilattice $A \times B$ will simply be a pair of truth values `(x . y)`, where x is a truth value in A and y is one in B. All of the bilattice operations on the product bilattice are computed pointwise using the corresponding operations on the component bilattices A and B.

The product construction is performed using the following function:

`product-bilattice(b1 b2)` Returns the bilattice that is the product of the bilattices b_1 and b_2.

14.2 Functional attachments

Another way in which to construct a bilattice is to start with a bilattice B and a function f that maps B into a fixed set S. The elements of the new bilattice are simply pairs of the form

```
(x . (f x))
```

where x is an element of B. The function f here is simply tagging along for the ride, and has no impact on the bilattice operations. It can be used to provide additional information about the truth values, and is used to compute probabilistic information in the bilattice **p** described in Section 13.2.4.

The relevant function is the following one:

`append-bilattice(b f)` Returns the bilattice obtained by appending the result of applying the function f to the points of the original bilattice b.

14.3 Surgeries

A somewhat more interesting construction is the one we described in Section 13.4 as a *surgery*. Here, we take one bilattice b_1 and replace the point corresponding to `unknown` in this bilattice with another bilattice b_2. The truth values in the resulting bilattice are either of the form `(T . x)` for a point x in the "upper" bilattice b_1, or `(NIL . y)` for a point y in the lower bilattice. Truth values from b_1 take priority over values from b_2.

To perform this construction generally, we have:

`splice-bilattice(upper lower)` Returns the bilattice obtained by replacing the point `unknown` in the bilattice `upper` with a copy of the bilattice `lower`.

As an example, the default bilattice **d** of Section 13.3 is given by

```
(splice-bilattice *first-order-bilattice* *first-order-bilattice*)
```

The circumscriptive bilattice of Section 13.4 is given by

```
(splice-bilattice *first-order-bilattice* *first-order-atms-bilattice*)
```

14.4 Hierarchies

It is possible to iterate the surgery construction of the previous section, producing a bilattice that is an infinite descending "chain" of copies of a basic bilattice b. This is done by the function `hierarchy-bilattice`:

`hierarchy-bilattice(b)` Returns the bilattice constructed as an infinite descending chain of copies of b.

Truth values in this bilattice have the form `(n . x)`, where x is an element of b, and n is an integer giving the hierarchy level. The top hierarchy level of the bilattice has $n = 0$; the second level is $n = 1$, and so on. The point `unknown` is given by `NIL`.

This function is used to construct the hierarchical version of the default bilattice.

14.5 Lattices

Bilattices can also be constructed from lattices. A lattice is a set equipped with a *single* partial order and greatest lower bound and least upper bound operations, which we denote by $+$ and $\cdot$ respectively.

An element of the associated bilattice is a *pair* of elements of the lattice, and the bilattice operations are given by:

$$
\begin{aligned}
\neg(a,b) &= (b,a) \\
(a,b)+(c,d) &= (a+c, b+d) \\
(a,b)\cdot(c,d) &= (a\cdot c, b\cdot d) \\
(a,b)\vee(c,d) &= (a+c, b\cdot d) \\
(a,b)\wedge(c,d) &= (a\cdot c, b+d)
\end{aligned}
$$

Roughly speaking, the first component of the bilattice element labels the "truth" of the sentence, and the second labels its falsity. This is how the ATMS bilattices are constructed.

The function used is `lattice-to-bilattice`:

`lattice-to-bilattice(l)` Returns the bilattice constructed from the lattice l.

A lattice is a structure rather like a bilattice, but with fewer slots (because it only has a single partial order). The fields in an instance of the `lattice` structure are:

`max` A LISP object that is the maximal element of the lattice.

`min` A LISP object that is the minimal element of the lattice.

`eq` A LISP function that takes two lattice elements as arguments and returns `T` if they are equal and `NIL` if they are not.

`and` A LISP function that takes two lattice elements as arguments and returns their greatest lower bound.

`or` A LISP function that takes two lattice elements as arguments and returns their least upper bound.

`leq` This optional field is a LISP function that accepts two arguments x and y and determines whether or not $x \leq y$ under the partial order corresponding to the lattice.

`vars` This LISP function accepts a lattice element as an argument and returns a list of the variables that it contains. The default value returns `NIL` independent of its argument.

`plug` This LISP function accepts as arguments a lattice value and a binding list, and returns a potentially modified lattice value. The default value is a function that leaves the lattice value unchanged.

`dws` This LISP function, "dot with star," accepts as arguments two lattice values, v_1 and v_2. The lattice element returned is v_1, modified so as to remove any contribution that could be attributed to the lattice element v_2.

`stash-val` The lattice element entering into the bilattice function `stash-val`.

`long-desc` A long description of the lattice, used for input and output only.

`short-desc` A short description of the lattice, also used for input and output only.

`char` A one-character identifier for the lattice.

14.6 Directed acyclic graphs

The final fashion in which MVL supports the construction of new bilattices is an extension of the product construction of Section 14.1. There, we formed the product of two bilattices $A \times B$. If the bilattices were identical, we would have formed the product bilattice $B \times B$, or B^2. Another way to view this bilattice is as the collection of functions from the two point set $\{1, 2\}$ into the original bilattice B. Since such a function f can be described as the pair $(f(1), f(2))$, there is no difference between this description and the previous one.

Given this analogy, it is not hard to see that we can extend the construction to the bilattice B^S for an arbitrary set S, where elements of this bilattice are functions from the set S into B, and all of the bilattice functions ($\wedge$, $\cdot$ and so on) are once again computed pointwise.

The difficulty with this idea is that for large sets S, it is not clear how we can represent a function from S into B in a convenient way. It turns out, however, that if S has the structure of a directed acyclic graph, or DAG, this problem can be overcome.

14.6.1 Defining the DAG

As a preliminary, we need some way to define the DAG itself. In MVL, a DAG is an instance of the `dag` structure; this structure has the following fields:

`root` The root of the DAG.

`eq` A function that accepts two elements of the DAG and returns `T` if they are the same element and `NIL` otherwise.

`leq` A function that accepts two elements of the DAG and returns `T` if the first is farther from the root of the DAG than the second. If not supplied by the user, this function can by computed from the `dot` operation on the DAG.

`dot` A function that accepts two elements of the DAG and returns a list of their first common descendants, or `NIL` if no such descendant exists.

`long` A long description of the DAG.

`short` A short description of the DAG.

`inherit` A function describing the fashion in which truth values are inherited within the DAG. This function f should take four arguments: a bilattice b, two elements of the DAG x and y, and an element z of the bilattice. If z is the element of the bilattice associated to the DAG element x, then $f(b, x, y, z)$ will be the value associated to the DAG element y in the absence of information to the contrary. By default, this function ignores its first three arguments and assigns y the same value as x has.

`vars` This LISP function accepts a DAG element as an argument and returns a list of the variables that it contains. The default value returns `NIL` independent of its argument.

`plug` This LISP function accepts as arguments a DAG value and a binding list, and returns a potentially modified DAG value. The default value is a function that leaves the DAG value unchanged.

14.6.2 Constructing the bilattice

Having selected a DAG D and a bilattice B, we can now construct the bilattice B^D. Specific functions from D to B are described as structures with the following slots:

`dag` The domain of the function, D.

`bilattice` The range of the function, B.

`list` A list of elements of the form `(p . v)`, where `p` is a point where the value changes in a fashion *other* than the one predicted by the inheritance function associated with D and `v` is the value taken there.

We will refer to these functions as DAG-*functions.*

There are a variety of functions available to manipulate DAG-functions, and these are used to construct the bilattice B^D. In some cases, however, the user may want to access these functions directly. Here are some of them:

`get-val(d f)` Returns the value of the function f at the point d of the DAG.

`make-tv(dag bilattice d b)` Returns a DAG-function that takes the value b at the point d. If d is not the root of the DAG, the value at the root of the DAG is generally the point `unknown` in the bilattice, but the user may supply the optional `root-fn` argument, in which case the value at the root of the DAG is the result of applying `root-fn` to the bilattice.

`make-root(dag bilattice b)` Returns a DAG-function that takes the value b at the root of the DAG.

`dag-change-tv(f b-f)` The argument `b-f` should be a function from the bilattice to itself; a modified version of the original DAG-function f is returned in which every point in the DAG-function has had the function `b-f` applied to it. There is an optional argument `simplify` that defaults to `NIL`; if `T`, the computed function is simplified before being returned. (See `simplify`.)

`dag-change-dag(f d-f)` The argument `d-f` should be a function from the DAG to itself; a modified version of the original DAG-function f is returned in which every point at which the DAG-function takes a new value has had the function `d-f` applied to it. The bilattice argument is expected to be the bilattice of functional truth values as opposed to the bilattice that is the target of the functions.

`dag-list-of-answers(f)` Given a DAG-function f, assumed to be from the DAG of binding lists into the currently active bilattice, this function constructs a list of answers corresponding to the points where the DAG-function takes values.

`dag-accumulate-fn(dag bilattice list)` Given a DAG, a bilattice, and a list of elements of the form

```
(dag-elt . bilattice-elt)
```

a DAG-function is constructed that takes the given values at the given elements of the DAG.

An optional argument to `dag-accumulate-fn` is a modifier function, which is used if the `d-list` contains the same DAG element more than once. In this case the modifier function is called with the new and old values (in that order) and the result is used as the value of the DAG-function at that particular point. The default action is to simply overwrite the old value with the new one. (In other words, the default modifier function simply returns its first argument while ignoring the second.)

`dag-prune(f pred)` Given a DAG-function f and a predicate on the bilattice, this function prunes from f all points d where $f(d)$ fails to satisfy the predicate.

`combine-fns(f1 f2 fn comp)` This function takes the two DAG-functions f_1 and f_2 and combines them using the function `fn`, which should be a function that takes two elements of the bilattice and returns a single one. The comparison function `comp` is used to determine if one of the two functions can be returned directly; if, for example `(comp f1 f2)` holds, than we simply return f_1, and similarly for f_2. As an example, if the combining function is the bilattice operation $\cdot$, the comparison function is `k-le`.

`combine-1(f type-info f1 ... fn)` This is an n-ary version of `combine-fns`, where f is an n-ary bilattice function and there are n DAG-functions supplied as well. The additional argument `type-info` indicates which of the n arguments to this function are not necessarily DAG-functions, but are instead parameters expected by the overall function f. This additional argument is a similar to the `args` field of a modal-op.

`add-dag(d b f)` Given the DAG-function f, add the value b at the point d. This function works by side effect only. There are two optional arguments: a modifier function that should take two elements of the bilattice as arguments and return a single one as a result. Given such a function m, the DAG-function is changed so that the new value at d is $m(b, f(d))$, where $f(d)$ is the old value. (This is similar to the optional argument in `dag-accumulate-fn`.) The second optional argument default to `T` and indicates that the resulting DAG-function should be simplified.

`simplify(f)` In some cases, manipulating the DAG-function f may cause it to contain redundancies, where the value at some point is presented explicitly even though this value is in fact the one that would normally be inherited from its parents. This function returns a modification of f where these irrelevant entries have been removed.

`fn-and(f1 f2)` This function conjoins the values of f_1 and f_2 using the bilattice $\wedge$ operation. The new DAG-function is returned.

`fn-or(f1 f2)` This function disjoins the values of f_1 and f_2 using the bilattice $\vee$ operation. The new DAG-function is returned.

`fn-plus(f1 f2)` This function adds the values of f_1 and f_2 using the bilattice $+$ operation. The new DAG-function is returned.

`fn-dot(f1 f2)` This function combines the values of f_1 and f_2 using the bilattice $\cdot$ operation. The new DAG-function is returned.

`fn-dws(f1 f2)` This function combines the values of f_1 and f_2 using the bilattice operation `dot-with-star`. The new DAG-function is returned.

`fn-not(f1)` This function negates the values of f_1 using the bilattice $\neg$ operation. The new DAG-function is returned.

`fn-eq(f1 f2)` This function compares the values of f_1 and f_2 using the bilattice comparison operation. Either `T` or `NIL` is returned.

`fn-comp(f1 f2 fn)` This functions compares the values of f_1 and f_2 using the bilattice comparison function `fn`. Either `T` or `NIL` is returned.

Of course, the point of all of this is to enable us to define the following function:

`dag-bilattice(dag bilattice)` Given a bilattice B and a dag D, returns the bilattice B^D.

The following functions are also available:

`lattice-to-dag(lattice)` Construct a DAG from the given lattice.

`lattice-to-dag-to-bilattice(lattice bilattice)` First construct a DAG D from the given lattice. Then use the bilattice B to construct the bilattice B^D.

14.6.3 The DAG of theories

Two specific DAGs are provided for use in the MVL system. The first uses a DAG corresponding to the existence of multiple theories within a single database; the second uses a DAG to handle binding information.

The theory DAG consists of a hierarchy of theories. In a single database, for example, we might want to have rules that applied to celestial bodies, rules applying to stars, rules applying only to red giants or only to our sun, and so on. The truth value of a rule applicable to all celestial bodies should be inherited if we are considering our sun (but it should be possible to overwrite it with a different truth value if we wish), but a rule applicable to red giants should not be inherited in this fashion. So we see that our DAG construction is well-suited for this sort of an application.

To activate the theory mechanism, simply invoke the LISP function activate-theory-mechanism. This function replaces the active bilattice, say B, with the bilattice constructed of functions from the theory DAG into B. The original bilattice can still be accessed, since it is the second component of the newly formed bilattice. The function theory-active-p can be used to determine whether or not the theory mechanism has been activated.

The DAG of theories is the value of the global variable theory-dag, and has as its root the theory whose name is the value of the constant *global*. To indicate that a theory big includes another theory little as a subtheory, invoke the function

```
(includes big little)
```

Thus, for example, one would introduce a new theory called theory by writing

```
(includes *global* 'theory)
```

The effects of includes can be reversed by invoking the function unincludes.

Note that since the theory DAG is used by the functions that manipulate the bilattice B^D, it is important that this DAG be maintained as new theories are introduced. If includes is not used to "locate" these new theories in the overall theory DAG itself, the theories will be inaccessible to many of the MVL functions.

At any point in time, any new fact will be inserted into the database as true in a particular theory; the theory into which it is inserted is the value of *active-theory*. The value of this variable should not be changed directly, but should instead be modified by invoking the function activate on the theory into which subsequent facts should be written. The variable true-in-theory contains the truth value of a sentence that is true in the currently active writing theory.

The theory *active-theory* is also the theory that is active for reading at any point in time. Of course, any parents of the active theory are also active by virtue of the structure of the theory DAG itself; if it is desired to read from multiple theories simultaneously, a new theory should be created that is a child of all of them.

Here is a summary of the functions described so far:

activate-theory-mechanism Activates the theory mechanism. Accepts as an optional argument either a one-character bilattice descriptor, or a bilattice.

theory-active-p Returns T or NIL depending on whether or not the theory mechanism has been activated.

theory-dag The DAG of theories.

global The root of the DAG of theories.

includes(big little) This function is used to state that the theory little is a subtheory of the theory big, and should inherit from it.

unincludes(big little) This function is used to state that the theory little is no longer a subtheory of the theory big, and should not inherit from it.

active-theory The theory that is currently active for reading and writing.

activate(th) Activate the theory th for reading and writing.

true-in-theory The truth value of a sentence that is true in the currently-active writing theory.

Finally, five functions have been added to simplify the user interface somewhat. The function theory-contents displays the contents of a given theory, and empty-theory removes all of the propositions in this theory. To modify the contents of a specific theory, the following functions can be used:

state-in-theory(p) Adds the proposition p to the currently active theory. Allowed keywords: **theory** can be used to specify a different theory to which p should be added; **value** and **increment-p** have the same meaning as for **state**. (See Section 3.3; if these keywords are not used, the function **state** can be used equally well, since the default stash value with the theory mechanism activated also inserts the proposition into the currently active theory.)

unstate-from-theory(p) Removes the proposition p from the currently active theory. Allowed keywords: **theory** can be used to specify a different theory from which p should be removed; **value** and **decrement-p** have the same meaning as for **unstate**. (See Section 3.3.)

erase-from-theory(p) Removes the proposition p from the currently active theory, and forward chains on the result. Allowed keywords: **theory** can be used to specify the theory from which p should be removed; **value**, **decrement-p** and **cutoffs** have the same meaning as for **erase**. (See Section 8.2.)

theory-contents() Displays the contents of the currently active theory, which can be changed using the keyword **theory**. The keywords **cutoffs** and **props** are allowed and are treated as for **contents**. (See Section 6.3.) There is an additional keyword, **this-theory-only**, which defaults to T and, if supplied, indicates that only sentences whose truth values *change* in the given theory are to be displayed. (This avoids displaying all sentences in the global theory, for example.)

empty-theory() Empties the currently active theory, although this theory may be changed using the keyword **theory**. The keyword **value** has a meaning similar to its meaning as a keyword to **erase**.

14.6.4 The DAG of binding lists

The other DAG supplied with the MVL system is used for binding lists since these, too, admit a natural inheritance structure. The root of the DAG is the empty binding list NIL, and inheritance in the DAG is determined by subsumption. The bilattice constructed as functions from this DAG into ***active-bilattice*** is used extensively by the prover, and is available to the user as **bdg-to-truth-val**.

The binding DAG itself is known as **binding-dag**, and is constructed using the following functions:

equal-binding(b1 b2) Checks to see if two binding lists are effectively the same.

append-binding-lists(b1 b2) Computes the least specific binding list that is more specific than both b_1 and b_2 or returns NIL if no such binding list exists (b_1 and b_2 may bind the same variable to different constants, for example).

binding-le(b1 b2) Returns NIL unless b_1 is a more specific binding that b_2, in which case a list of relative binding lists is returned.

The following global variables are also used:

binding-dag The DAG of binding lists.

bdg-to-truth-val The bilattice of functions from the DAG of binding lists into ***active-bilattice***.

15 Modal operators

An additional feature that makes the MVL system unique is its ability to handle *modal operators*. The approach taken by MVL to modal operators is that they are functions from the bilattice of truth values to itself. Thus, for example, the modal operator that takes a sentence p into the sentence, "I know that p," corresponds to the bilattice function (on the first-order bilattice) that takes t and $\perp$ into t (since we know p if we know it to be true or know it to be true and false both), and takes f and u into f (since we do *not* know p if we either know it to be false or do not know if it is true or false). Further details can be found in the paper, "Bilattices and modal operators."

A modal expression in MVL is any sentence of the form

```
(op p-1 ... p-n)
```

where `op` is recognizable as a modal operator because it has a non-`NIL` value for its `modal-op` property in the currently active invocation of MVL. In this case, the value of the `modal-op` property should be an instance of the structure `modal-operator`. This structure has the following fields:

`name` The name of the modal operator. This is used to identify modal operators in different bilattices that have the same commonsense meaning (and can therefore be usefully combined when the bilattices are combined as in the previous section).

`fn` A function that accepts as arguments the truth values of the sentences under the scope of the modal operator and returns the truth value of the modal expression itself.

`args` If the modal operator accepts n arguments, this should be a list of n entries, each of which is given a value as follows:

- **0** The argument is a parameter and not a sentence on which the prover should be invoked.
- `T` The argument is a sentence; only the sentence itself need be considered by the prover.
- `NIL` The argument is a sentence; only the negation need be considered by the prover.
- **1** The argument is a sentence; the sentence should be considered by the prover if the enclosing modal expression is used positively in a proof, and negatively otherwise.
- **-1** The reverse of the previous case.
- **x** Both the sentence and its negation should be considered by the prover.

As an example, suppose that we are working with a bilattice where the truth values are functions describing the behaviors of various fluents over the course of time, and have a modal operator `delay`, where the truth value assigned to

```
(delay t p)
```

is expected to be the same as the truth value assigned to p but delayed by an amount of time t. Here, the first argument (t) is a parameter and the second argument (p) is not; the second argument matches the sense of the modal expression and the `args` for `delay` are therefore `(0 1)`.

For the modal operator L of knowledge, however, there is a single argument and all we ever need to know is if the argument can be proven from the information in the database. The `args` for L are therefore simply `(T)`.

All parametric arguments should be bound when attempting to derive a modal expression. This is similar to the requirement that arguments to a procedurally-attached predicate be bound when the procedure is called.

15.1 Modal operators supplied with the MVL bilattices

All bilattices in MVL are equipped with a modal operator `idn` that does not change the truth value of its argument at all; this modal operator can be used to semantically mark points where it may be desirable to interrupt the inference process.

The bilattices supplied as part of the MVL system are equipped with the following modal operators:

`*first-order-bilattice*` This bilattice has two modal operators: L, which maps p into, "necessarily p," or, "I know that p," and M, which maps p into "possibly p."

`*atms-bilattice*` No modal operators are supplied with this or any of the other ATMS-type bilattices.

`*default-bilattice*` This bilattice and the hierarchical default bilattice both inherit the modal operators L and M from the first-order bilattice.

`*circumscription-bilattice*` This bilattice does not come equipped with any modal operators.

`*time-bilattice*` This bilattice inherits the modal operators L and M from the bilattice of basic values. The following three additional modal operators are also defined:

1. `delay(t p)` This is a modal operator that pushes the truth value of p into the future by an amount of time t. Thus if p is true at time 2, $\mathtt{delay}(p, 2)$ will be true at time 4.
2. `propagate` is a modal operator that corresponds to the frame axiom. If p is true at time 2, unknown between times 3 and 20, and false at time 21, then $\mathtt{propagate}(p)$ will be true from times 2 until 20 and false subsequently.
3. `at(t p)` is a modal operator that returns the value taken by the truth value p at the time t. (In actuality, a function has to be returned; the function is the one that takes value $p(t)$ at all times instead of just at t.)

`*plan-bilattice*` The planning bilattice has the same modal operators as does the temporal reasoning bilattice described just above: L, M, `delay`, `propagate` and `at`. The parametric argument to `delay` continues to be an integer (the number of actions that could transpire during the delay); the parametric argument to `at` is a plan (at which the truth value should be evaluated).

15.2 Quantification and modal operators

In addition to `idn`, there are two other modal operators that are available for use with any bilattice, `exists` and `forall`. Although not really modal operators on the bilattice being used, these *are* legitimate modal operators on the bilattice `bdg-to-truth-val` that was described in Section 14.6.4.

The functions associated to these modal operators mimic the usual behaviors given to the quantifiers. Thus an expression such as

`(exists ?x (foo ?x))` (6)

is evaluated by first computing the truth value that could be assigned to `(foo ?x)` as an element of `bdg-to-truth-val`, so that a truth value is obtained for every binding of `?x`. This truth value is then manipulated so that the values for any particular binding of `?x` are disjoined (the usual interpretation of the existential quantifier), and the result is returned as the truth value assigned to (6). The modal operator `forall` is handled similarly and corresponds to universal quantification. The first (parametric) argument to these operators can also be a list of variables, in which case all of the variables are quantified over.

15.3 Modal operators created automatically

Finally, many of the functions that construct new bilattices are able to construct modal operators on these bilattices in terms of the modal operators on the bilattices used in the construction. More specifically:

1. A product bilattice $A \times B$ will inherit a modal operator from A and B if these component bilattices have modal operators with the same name.
2. A bilattice constructed using `append-bilattice` will have modal operators corresponding to whatever modal operators were available on the bilattice used in the construction.
3. A bilattice constructed by splicing two bilattices A and B together will inherit a modal operator from A and B if these component bilattices have modal operators with the same name.
4. A bilattice constructed using `hierarchy-bilattice` will have modal operators corresponding to whatever modal operators were available on the bilattice used in the construction.

5. A bilattice constructed from a lattice will not normally have any modal operators associated to it.
6. The bilattice B^D will inherit whatever modal operators the bilattice B has.

Acknowledgement

Support for the development of MVL has been provided by the Air Force Office of Scientific Research, the Defense Advance Research Projects Agency, General Dynamics, the National Science Foundation, Rockwell International, and Rome Labs.

A Summary of MVL functions and constants

activate(th) Activate the theory th for reading and writing.

activate-theory-mechanism Activates the theory mechanism. Accepts as an optional argument either a one-character bilattice descriptor, or a bilattice.

active-bilattice The currently active bilattice.

active-theory The theory that is currently active for reading and writing.

add-dag(d b f) Given the DAG-function f, add the value b at the point d. This function works by side effect only. There are two optional arguments: a modifier function that should take two elements of the bilattice as arguments and return a single one as a result. Given such a function m, the DAG-function is changed so that the new value at d is $m(b, f(d))$, where $f(d)$ is the old value. (This is similar to the optional argument in `dag-accumulate-fn`.) The second optional argument default to T and indicates that the resulting DAG-function should be simplified.

answer-binding(a) Returns the binding field of the answer a.

answer-value(a) Returns the value field of the answer a.

anytime-trace(pat) Include pat on *anytime-trace*, so that any instance of pat will be traced. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

anytime-untrace(pat) Remove any instance of pat from *anytime-trace*. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

append-bilattice(b f) Returns the bilattice obtained by appending the result of applying the function f to the points of the original bilattice b.

append-binding-lists(b1 b2) Computes the least specific binding list that is more specific than both b_1 and b_2 or returns NIL if no such binding list exists (b_1 and b_2 may bind the same variable to different constants, for example).

atms-bilattice The bilattice corresponding to an ATMS, where the justification information is maintained in disjunctive normal form as suggested by de Kleer.

(bagof ?x ?p ?b) This relation holds for x, p and b if and only if b is a list of all x satisfying p.

bc(q) Attempts to prove the query q from information in the database, looking only for a single answer.

bc-trace(pat) Include pat on *bc-trace*, so that any instance of pat will be traced. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

bc-untrace(pat) Remove any instance of pat from *bc-trace*. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

bc-unwatch(preds) Removes the given predicates from the list of WATCHed predicates. If no arguments are supplied, all predicates are unWATCHed.

`bc-watch(preds)` Accepts as arguments any number of predicates that should be watched as the prover proceeds. Each predicate can also be a list of the form `(predicate depth)` where `depth` is the number of successive ancestors to be displayed when the prover encounters an instance of the predicate. Returns a list of the predicates currently being WATCHed.

`bcs(q)` Attempts to prove the query q from information in the database, looking for all possible answers.

`bdg-to-truth-val` The bilattice of functions from the DAG of binding lists into `*active-bilattice*`.

`bilattice(b)` Add the bilattice b to `*bilattices*`.

`*bilattices*` A list of the bilattices known to the system.

`binding-dag` The DAG of binding lists.

`binding-le(b1 b2)` Returns `NIL` unless b_1 is a more specific binding that b_2, in which case a list of relative binding lists is returned.

`bottom` The truth value `bottom` in the active bilattice.

`*break-node*` The number of a node at which the proof process should be interrupted.

`choose-proof-and-tasks` Returns two values: a proof effort, and one or more tasks that use it. This proof effort is examined next, and any answers found are passed to the given tasks. The current implementation of MVL simply selects the shallowest active task.

`circum(p)` Like `bc`, but looks for instances of p that follow from the circumscription axiom. Allowed keywords: `cutoffs` and `succeeds`.

`*circum-trace*` If not `NIL`, the circumscriptive proof functions will output diagnostics describing their progress.

`circums(p)` Like `bcs`, but looks for instances of p that follow from the circumscription axiom. Allowed keywords: `cutoffs` and `succeeds`.

`*circumscription-bilattice*` The bilattice used to compute the results of the circumscription axiom.

`cnf(p)` Returns p in conjunctive normal form, as a list of conjuncts, where each conjunct is a list of disjuncts. Thus

```
(CNF '(AND A B))
```

returns

```
((A) (B))
```

`*cnf-atms-bilattice*` An alternative version of the ATMS bilattice where the justification information is maintained in conjunctive normal form.

`combine-1(f type-info f1 ... fn)` This is an n-ary version of `combine-fns`, where f is an n-ary bilattice function and there are n DAG-functions supplied as well. The additional argument `type-info` indicates which of the n arguments to this function are not necessarily DAG-functions, but are instead parameters expected by the overall function f. This additional argument is a similar to the `args` field of a modal-op.

`combine-fns(f1 f2 fn comp)` This function takes the two DAG-functions f_1 and f_2 and combines them using the function `fn`, which should be a function that takes two elements of the bilattice and returns a single one. The comparison function `comp` is used to determine if one of the two functions can be returned directly; if, for example `(comp f1 f2)` holds, than we simply return f_1, and similarly for f_2. As an example, if the combining function is the bilattice operation $\cdot$, the comparison function is `k-le`.

`compile-mvl` Compiles and loads the MVL system. On a Symbolics system, the keyword `reload` forces all of the files to be reloaded and the keyword `recompile` forces them all to be recompiled.

`*completed-database*` If `T`, find all exceptions to default proofs. If `NIL` (the default), accept defaults proofs with possible exceptions as complete.

`cont-analysis(analysis)` This function continues the investigation of a specific analysis, and can be used to resume a proof effort that has been curtailed for some reason. It returns two values: an instance of the `answer` structure giving the results of the proof attempts, and a modified version of the `analysis` structure that can be used to continue the proof process if desired.

`cont-prover(proof)` Resume the task associated to the auxiliary information in the given instance of the `proof` structure. This function returns a list of `bcanswers` corresponding to the result obtained by the prover, and returns as soon as a solution to the original query is found.

`contents` Displays the contents of the database. Keywords: `cutoffs`, `props`.

`create-mvl-invocation` This function sets up a new copy of MVL. It accepts an optional argument that is an instance of `mvl-invocation`; this can be used to reset the state of MVL to a previous one. If the optional argument is omitted, default values are used. The value returned is the MVL invocation that was active before the function was called, so that another call to `create-mvl-invocation` can be used to restore the original environment.

`cutoffs-not(cutoffs &optional bilattice)` Returns a new cutoffs c such that

```
(test x c) = (test (mvl-not x) cutoffs)
```

for all truth values x. In other words, x passes c if and only if $\neg x$ passed the cutoffs supplied originally.

`dag-accumulate-fn(dag bilattice list)` Given a DAG, a bilattice, and a list of elements of the form

```
(dag-elt . bilattice-elt)
```

a DAG-function is constructed that takes the given values at the given elements of the DAG.

An optional argument to `dag-accumulate-fn` is a modifier function, which is used if the `d-list` contains the same DAG element more than once. In this case the modifier function is called with the new and old values (in that order) and the result is used as the value of the DAG-function at that particular point. The default action is to simply overwrite the old value with the new one. (In other words, the default modifier function simply returns its first argument while ignoring the second.)

`dag-bilattice(dag bilattice)` Given a bilattice B and a dag D, returns the bilattice B^D.

`dag-change-dag(f d-f)` The argument `d-f` should be a function from the DAG to itself; a modified version of the original DAG-function f is returned in which every point at which the DAG-function takes a new value has had the function `d-f` applied to it. The bilattice argument is expected to be the bilattice of functional truth values as opposed to the bilattice that is the target of the functions.

`dag-change-tv(f b-f)` The argument `b-f` should be a function from the bilattice to itself; a modified version of the original DAG-function f is returned in which every point in the DAG-function has had the function `b-f` applied to it. There is an optional keyword `simplify` that defaults to `NIL`; if `T`, the computed function is simplified before being returned. (See `simplify`.)

`dag-list-of-answers(f)` Given a DAG-function f, assumed to be from the DAG of binding lists into the currently active bilattice, this function constructs a list of answers corresponding to the points where the DAG-function takes values.

`dag-prune(f pred)` Given a DAG-function f and a predicate on the bilattice, this function prunes from f all points d where $f(d)$ fails to satisfy the predicate.

`*default-bilattice*` A bilattice that can be used for simple default reasoning.

`delete-bdg(var bdg-list)` Destructively modifies `bdg-list`, removing any binding for the variable `var`.

`demo-mvl` Demonstrate the MVL system by invoking the function `mvl-demo-file` on the file `MVL.DEMO`.

`denotes(a)` Returns the proposition associated with a particular LISP atom.

`deselect-tasks(tasks)` Deselect the given tasks (used by the graphical display tools).

`dnf(p)` Returns p in disjunctive normal form, as a list of disjuncts, where each disjunct is a list of conjuncts. Thus

```
(DNF '(AND A B))
```

returns

```
((A B))
```

`(done exp-1 ... exp-n)` This predicate, which is useful for stashing only, causes LISP to evaluate each of the subsequent expressions.

`dot-with-not(x)` Computes $x \cdot \neg x$.

`dot-with-star` The function `dws` in the active bilattice.

`dynamic-unview()` Instructs MVL to cease using the dynamic viewer when the prover is invoked.

`dynamic-view()` Instructs MVL to use the dynamic viewer to show the progress of the prover.

`empty` Unstashes every sentence in the database. Keyword: `value`.

`empty-theory()` Empties the currently active theory, although this theory may be changed using the optional theory argument. The keyword `value` has a meaning similar to its meaning as a keyword to `erase`.

equal-answer(a1 a2) Returns T if the two answers a1 and a2 are equal, and NIL otherwise. The functions used are binding-equal and mvl-eq, the functions normally used to test binding lists and truth values for equality.

equal-binding(b1 b2) Checks to see if two binding lists are effectively the same.

equivalence-translation Controls the fashion in which normal-form deals with the connective <=>.

erase(p) Removes p from the database and forward chains. Allowed keywords: value, decrement-p and cutoffs.

erase-from-theory(p) Removes the proposition p from the currently active theory, and forward chains on the result. Allowed keywords: theory can be used to specify the theory from which p should be removed; value, decrement-p and cutoffs have the same meaning as for erase. (See Section 8.2.)

existing-mvl-invocation This function returns a single value that is an instance of the structure mvl--invocation. It is useful primarily for saving the current copy of MVL so that it can be restored later.

false The truth value false in the active bilattice.

fc(p) Inserts the sentence p into the database and forward chains. Allowed keywords: value, increment-p, fc-increment-p, cutoffs and include-modal.

fc-trace(pat) Include pat on *fc-trace*, so that any instance of pat will be traced. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

fc-untrace(pat) Remove any instance of pat from *fc-trace*. The argument is optional; if omitted, (?*) (of which everything is an instance) will be used.

first-order-atms-bilattice This bilattice corresponds to a fully first-order ATMS, as opposed to the more conventional one dealing only with propositional logic.

first-order-bilattice The bilattice corresponding to first-order logic.

fn-and(f1 f2) This function conjoins the values of f_1 and f_2 using the bilattice $\wedge$ operation. The new DAG-function is returned.

fn-comp(f1 f2 fn) This functions compares the values of f_1 and f_2 using the bilattice comparison function fn. Either T or NIL is returned.

fn-dot(f1 f2) This function combines the values of f_1 and f_2 using the bilattice $\cdot$ operation. The new DAG-function is returned.

fn-dws(f1 f2) This function combines the values of f_1 and f_2 using the bilattice operation dot-with-star. The new DAG-function is returned.

fn-eq(f1 f2) This function compares the values of f_1 and f_2 using the bilattice comparison operation. Either T or NIL is returned.

fn-not(f1) This function negates the values of f_1 using the bilattice $\neg$ operation. The new DAG-function is returned.

fn-or(f1 f2) This function disjoins the values of f_1 and f_2 using the bilattice $\vee$ operation. The new DAG-function is returned.

`fn-plus(f1 f2)` This function adds the values of f_1 and f_2 using the bilattice + operation. The new DAG-function is returned.

`get-bdg(var bdg-list)` Returns the value assigned to the variable `var` by the binding list `bdg-list`. If the variable does not appear in the binding list, `NIL` is returned.

`get-mvl (symbol indicator)` Like the Common Lisp `get`, but if the value is a list, treats it as an a-list and searches for an item whose car matches `*mvl-invocation*`. The value taken can be changed with `setf`.

`get-val(d f)` Returns the value of the function f at the point d of the DAG.

`*global*` The root of the DAG of theories.

`groundp(p)` Returns `T` if the proposition p is ground (i.e., contains no variables) and `NIL` otherwise.

`hierarchy-bilattice(b)` Returns the bilattice constructed as an infinite descending chain of copies of b.

`*hierarchy-bilattice*` A bilattice similar to the default bilattice but where the defaults are labelled with integer-valued strengths.

`*if-translation*` Controls the fashion in which `normal-form` deals with the connective `if`.

`includes(big little)` This function is used to state that the theory `little` is a subtheory of the theory `big`, and should inherit from it.

`index(p)` Updates the discrimination net as needed to insert the proposition p into the database.

`init-analysis(prop)` The MVL primitive upon which the proving functions are based, this function accepts a proposition and creates and returns an instance of the `analysis` structure.

`init-prover(p)` This function initializes and returns an instance of `proof` containing auxiliary information about the proof process.

`instp(p1 p2)` If p_2 is an instance of p_1, return a list of their most general unifiers; only variables in p_1 are bound. Returns `NIL` if p_2 is not an instance of p_1.

`invoke(fn p)` This function invokes `fn` on the sentence p. In addition, any keyword arguments are passed to `fn`, but with `&allow-other-keys` so that the keywords will be ignored if `fn` is not expecting them.

`k-ge(x y)` Returns `T` if $x \geq_k y$, and `NIL` otherwise.

`k-gt(x y)` Returns `T` if $x >_k y$, and `NIL` otherwise.

`k-le(x y)` Returns `T` if $x \leq_k y$, and `NIL` otherwise.

`k-lt(x y)` Returns `T` if $x <_k y$, and `NIL` otherwise.

`k-not-ge(x y)` Returns `T` if $x \not\geq_k y$, and `NIL` otherwise.

`k-not-gt(x y)` Returns `T` if $x \not>_k y$, and `NIL` otherwise.

`k-not-le(x y)` Returns `T` if $x \not\leq_k y$, and `NIL` otherwise.

`k-not-lt(x y)` Returns `T` if $x \not<_k y$, and `NIL` otherwise.

`(known p)` Stashing this predicate simply stashes p; unstashing it causes an error. Looking it up looks up p.

`lattice-to-bilattice(l)` Returns the bilattice constructed from the lattice l.

`lattice-to-dag(lattice)` Construct a DAG from the given lattice.

`lattice-to-dag-to-bilattice(lattice bilattice)` First construct a DAG D from the given lattice. Then use the bilattice B to construct the bilattice B^D.

`lisp-defined(fn)` This macro is used to indicate that the given function can be evaluated by a LISP call. It is also used to cause MVL to signal an error if an attempt is made to stash or unstash an instance of the function. `Lisp-defined` also accepts an optional argument, which is an inverter function. The overall result of attempting to lookup an instance of the function can be described as follows:

1. If none of the arguments to the function (i.e., all but the last of the arguments of the relation) is a variable, then the function is called and the result is unified with the final argument of the relation.
2. If the inverter function has been supplied and is equal to `T`, then the function is called and the result is unified with the final argument of the relation even if some of the arguments are variables.
3. If some of the arguments are variables but the result is not a variable, then the inverter function is invoked if one has been supplied. The arguments passed to the inverter function are the result (i.e., the last argument of the relation), and a list of all of the arguments (i.e., all but the last argument of the relation). If there is no inverter function, then `NIL` is returned. In addition, if `*warn-on-bad-lisp-invocations*` is not `NIL`, the user is warned.
4. If both some of the arguments and the result are variables, then `NIL` is returned. In addition, if `*warn-on-bad-lisp-invocations*` is not `NIL`, the user is warned.

`lisp-predicate(pred)` This macro is used to indicate that the given predicate can be evaluated by a LISP call. It is also used to cause MVL to signal an error if an attempt is made to stash or unstash an instance of the function.

`load-logic` Empty the database and make a new bilattice active. Accepts as an optional argument either a character designating a bilattice, or the new bilattice itself. If no argument is supplied, the user is prompted for the new bilattice.

`load-mvl` Loads the MVL system.

`lookup(q)` Searches the database for a sentence matching the query q. Allowed keywords: `cutoffs`, `inv`, `allow-procedural-attachment`, `succeed-on-modal-ops`. Returns two values: an answer and the source of the match. Returns `NIL` if the query cannot be found in the database.

`lookup-call(fn p)` This function is similar to `invoke`, but modifies the values returned to make them agree with those expected to be returned by `lookup`. Unless the function `fn` supplies them, the truth value is assumed to be `true`. The second value is `attach`.

`lookups(q)` Similar to `lookup`, but looks for *all* sentences matching the query. Returns two lists of values.

`lookups-call(fn p)` The plural version of `lookup-call`.

`make-root(dag bilattice b)` Returns a DAG-function that takes the value b at the root of the DAG.

`make-tv(dag bilattice d b)` Returns a DAG-function that takes the value b at the point d. If d is not the root of the DAG, the value at the root of the DAG is generally the point `unknown` in the bilattice, but the user may supply the optional `root-fn` argument, in which case the value at the root of the DAG is the result of applying `root-fn` to the bilattice.

`matchp(pattern atom)` Returns a list of elements of the form `(b . d)` where b is a most general binding list such that `(plug pattern b)` is an instance of the proposition labelled with `atom` and d is a binding list for the variables in the database proposition. `NIL` is returned if no such binding list exists. Thus if `P1` is `(TALL #:?2)`, and `P2` is `(TALL TOM)`, then

```
(matchp (TALL TOM) P1)
```

returns a list consisting of the single entry

```
(NIL . ((#:?2 . TOM)))
```

The `car` of this entry is `NIL`, indicating that no variables in the query need to be bound to match `P1`; the `cdr` is `((#:?2 . TOM))` since the variable `#:?2` in `P1` needs to be bound to `TOM` to match the query.

```
(matchp (TALL ?X) P2)
```

returns `((((?X . TOM)) . NIL))`.

`mvl-and` The function `and` in the active bilattice.

`mvl-demo-file(string)` Can be used to construct new demos or tutorials.

`mvl-dot` The function `dot` in the active bilattice.

`mvl-eq` The function `eq` in the active bilattice.

`mvl-file(string)` Returns the path for the MVL file associated with the given name. The file name may include an extension, one may be supplied as an optional argument, or `.MVL` will be used by default. There is an optional keyword `no-error` indicating that even if the file cannot be found, the path should be returned.

`*mvl-invocation*` A global variable containing a unique identifier for the currently active copy of MVL.

`mvl-load(file)` Loads the sentences in the given file into the MVL database.

`mvl-not` The function `not` in the active bilattice.

`mvl-or` The function `or` in the active bilattice.

`mvl-plug` The function `plug` in the active bilattice.

`mvl-plus` The function `plus` in the active bilattice.

`mvl-print` This function accepts a single argument, a truth value, and modifies it in a way that will cause it to be pretty-printed. The value returned by mvl-print should not be used for any purpose other than printing.

`mvl-save(file)` Writes the MVL database out to the given file. This function also accepts the keywords `cutoffs` and `props`, which are treated as for `contents` (see Section 6.3). Finally, the keyword `if-exists` is passed to the `with-open-file` macro.

`mvl-test-file(string)` Can be used to evaluate various forms by the MVL system and compare the results to predefined values. Both the forms and the expected answers are in the `.TEST` file whose name is supplied to the function. Returns the number of errors found.

`mvl-unload(file)` Removes the sentences in the given file from the MVL database.

`mvl-vars` The function `vars` in the active bilattice.

`negate(p)` Returns the negation of p.

`new-*var()` Returns a new temporary sequence variable.

`new-?var()` Returns a new temporary variable.

`new-val(a v)` Sets the truth value of the sentence associated with the atom a to v. If v is the truth value `unknown`, the sentence is removed from the database.

`normal-form(p)` Returns an equivalent form of p suitable for insertion into the database (i.e., with the connectives `IFF`, `IF` and `<=>` reduced to `<=` and `=>`).

`*overviewer-size*` A list of the form (width height), this specifies the initial size of the overviewer window.

`plug(p bdg-list)` Returns the result of instantiating the proposition p using the given binding list. This function is nondestructive.

`prfacts(atom)` Displays the database facts concerning the given atom. Keyword: `cutoffs`.

`*probability-bilattice*` An extended version of the ATMS bilattice that maintains probabilistic information as well.

`product-bilattice(b1 b2)` Returns the bilattice that is the product of the bilattices b_1 and b_2.

`(property symbol value indicator)` Stashing this predicate sets the symbol's `indicator` property to `value` for the current invocation of MVL. Unstashing it removes the `indicator` property for the current invocation of MVL if the previous value of this property unifies with `value`. (See the definitions of `get-mvl` and `rem-mvl`.)

`props()` Returns a list of all propositions about which any information has been recorded.

`release-mvl` (Symbolics only) Compiles and loads the MVL system, and declares the new version to be `released`. The keyword `no-recompile` can be used to skip the compilation step.

`rem-mvl (symbol indicator)` Like the Common Lisp `remprop`, but if the value is a list, treats it as an a-list and removes the item whose car matches `*mvl-invocation*`.

samep(p1 p2) — Returns T if the propositions p_1 and p_2 are identical (up to variable renaming), and NIL otherwise. If the optional return-answer? argument is supplied and the propositions are the same, a binding list is returned instead of T.

save-nodes — If T (the default), nodes are labelled with consecutive integers when they are displayed by the first-order prover.

select-tasks(tasks) — Select the given tasks (used by the graphical display tools).

selected-tasks() — Returns a list of all selected tasks (used by the graphical display tools).

set-depth-limit — This function accepts a single optional argument and sets *depth-limit* to the given value. The default is NIL.

simplify(f) — In some cases, manipulating the DAG-function f may have cause it to contain redundancies, where the value at some point is presented explicitly even though this value is in fact the one that would normally be inherited from its parents. This function returns a modification of f where these irrelevant entries have been removed.

splice-bilattice(upper lower) — Returns the bilattice obtained by replacing the point unknown in the bilattice upper with a copy of the bilattice lower.

standardize-operators(p) — Returns an equivalent form of p with the connectives <=>, IF and IFF removed. This function does not use the control variables *if-translation* and *equivalence-translation*, since it is used when proving sentences, not when inserting them into the database.

stash(p) — Inserts p into the database. The allowed keywords are value, increment-p and allow-procedural-attachment.

stash-value — The function stash-val in the active bilattice.

state(p) — Inserts p into the database after converting it to normal form. Keywords: value, increment-p.

state-in-theory(p) — Adds the proposition p to the currently active theory. Allowed keywords: theory can be used to specify a different theory to which p should be added; value and increment-p have the same meaning as for state. (See Section 3.3; if these keywords are not used, the function state can be used equally well, since the default stash value with the theory mechanism activated also inserts the proposition into the currently active theory.)

static-display(a) — Invoke the static viewer, where a is the analysis structure returned by MVL when the proof is complete.

std-cutoffs — Cutoff information constructed from the value of cutoff-val in the active bilattice. See Section 12.2.2.

t-ge(x y) — Returns T if $x \geq_t y$, and NIL otherwise.

t-gt(x y) — Returns T if $x >_t y$, and NIL otherwise.

t-le(x y) — Returns T if $x \leq_t y$, and NIL otherwise.

t-lt(x y) — Returns T if $x <_t y$, and NIL otherwise.

t-not-ge(x y) — Returns T if $x \not\geq_t y$, and NIL otherwise.

t-not-gt(x y) Returns T if $x \not>_t y$, and NIL otherwise.

t-not-le(x y) Returns T if $x \not\leq_t y$, and NIL otherwise.

t-not-lt(x y) Returns T if $x \not<_t y$, and NIL otherwise.

test(value cutoffs &optional bilattice) Returns T if the truth value passes the test associated to **cutoffs**, and NIL otherwise. The test is performed for the supplied bilattice, or for the currently active one is the optional argument is omitted.

test-mvl Tests the MVL system by invoking the function **mvl-test-file** on the file MVL.TEST.

theory-active-p Returns T or NIL depending on whether or not the theory mechanism has been activated.

theory-contents() Displays the contents of the currently active theory, which can be changed using the keyword **theory**. The keywords **cutoffs** and **props** are allowed and are treated as for **contents**. (See Section 6.3.) There is an additional keyword, **this-theory-only**, which defaults to T and, if supplied, indicates that only sentences whose truth values *change* in the given theory are to be displayed.

theory-dag The DAG of theories.

time-bilattice A compound bilattice that can be used for reasoning about events occurring against a temporal background.

true The truth value **true** in the active bilattice.

true-in-theory The truth value of a sentence that is true in the currently-active theory.

truth-value(a) Returns the truth value associated with a particular LISP atom. Thus, if the result of asserting **(not (short tom))** into the database was the atom P38, **(denotes 'p38)** would evaluate to **(short tom)**, and **(truth-value 'p38)** would evaluate to **false**.

unifyp(p1 p2) Returns a list of the most general unifiers of p_1 and p_2, or NIL if the propositions fail to unify.

unincludes(big little) This function is used to state that the theory **little** is no longer a subtheory of the theory **big**, and should not inherit from it.

unindex(p) Removes from the discrimination net information relating to the proposition p.

unknown The truth value **unknown** in the active bilattice.

(unknown p) Stashing this predicate unstashes p; unstashing it causes an error. Looking it up fails if p can be found in the database, and succeeds otherwise.

unstash(p) Removes p from the database. Allowed keywords: **value**, **decrement-p** and **allow-procedural-attachment**.

unstash-facts-with-atom(atom) Unstashes from the database any facts concerning the given atom. Keywords: **value**, **decrement-p**, **allow-procedural-attachment**.

unstash-value The function **unstash-val** in the active bilattice.

unstate(p) Removes p from the database after converting it to normal form. Keywords: **value**, **decrement-p**.

unstate-from-theory(p) Removes the proposition p from the currently active theory. Allowed keywords: `theory` can be used to specify a different theory from which p should be removed; `value` and `decrement-p` have the same meaning as for `unstate`. (See Section 3.3.)

(value symbol val) Stashing this predicate sets the given symbol to the value `val`. Unstashing it makes the symbol unbound if its current value unifies with `val`.

varp(x) Returns `T` if x is a variable, and `NIL` otherwise.

varp*(x) Returns `T` if x is a sequence variable, and `NIL` otherwise.

varp?(x) Returns `T` if x is a standard (i.e., non-sequence) variable, and `NIL` otherwise.

vars-in(p) Returns a list of the variables appearing in p.

vartype(x) Returns ? if x is a standard variable, $*$ if x is a sequence variable, and `NIL` is x is not a variable.

viewer-scroll-bars When set to `T`, the viewer's scroll bars become active and can be used. The default value is `NIL` since the overviewer provides a better way to move around the tree. This variable has to be set before the viewer window is created in order to have any effect.

viewer-size A list of the form (width height), this specifies the initial size of the viewer window.

with-mvl-invocation This is a macro similar to the Common Lisp macro `with-open-file`. It accepts an optional argument that is an instance of `mvl-invocation` and is followed by a body of forms (presumably including a `load-logic`) that are evaluated in a locally-bound copy of MVL.

Index