The Framework for Resolution-based Automated Proof Procedure Systems Frapps 2.0 Users' Guide *

Michael K. Mitchell, Tomás E. Uribe and Alan M. Frisch

University of Illinois at Urbana-Champaign
Department of Computer Science and Beckman Institute
405 North Mathews
Urbana, Illinois 61801

September 28, 1991

1 Introduction

The Framework for Resolution-based Automated Proof Procedure Systems (Frapps), developed by Michael Mitchell and Tomás Uribe under the direction of Alan Frisch, is a package of functions written in Common Lisp that provide a set of building blocks from which resolution-based deductive systems can be easily constructed. Enough high-level functionality is provided to create a framework that should free users from low-level concerns, allowing them to focus both thought and energy on the "creative" issues of such a task.

Frapps operates by growing a *derivation graph*, into which clauses are integrated as *nodes*. The derivation graph is constructed, accessed and maintained by the functions described within this document. The basic organization of the derivation graph is by *levels*: Once user-defined clauses are integrated as nodes at level 0, resolvents and factors can be formed by applying inference rules to previously integrated clauses (i.e. nodes), and can then be integrated themselves. In this way new levels in the graph are created, until a given termination condition is reached. Various operations such as listing, deactivating and reactivating nodes throughout the derivation graph can be performed. A *cost-function priority queue*, maintained by the system as the derivation proceeds, is provided as the basis for implementing complex search strategies.

*This is an expanded and revised version of the original *FRAPPS Users' Guide*, M.K. Mitchell, 1989.

Several inference rules and numerous deletion and search strategies are provided to facilitate the construction of a wide variety of "tailored" resolution systems. Such a system will involve writing new procedures based on the FRAPPS functions described in this manual.

FRAPPS also provides several *session management* procedures that allow the user to save the *state* of a current session and later resume it from where it was previously stopped. This feature is particularly useful during the course of a "user-guided" proof.

1.1 The Basic Search Algorithm

The following is a schematic outline of a search procedure as most easily implemented using FRAPPS:

1. The initial set of clauses (the "base-set") is integrated as the set of nodes at level 0 in the derivation graph. This is usually a set of axioms and the negation of the proposition to be proved (under these axioms), written in clausal form.
2. Nodes in the derivation graph can be *active* or *inactive*. When each new clause is integrated (as an active node), the set of all other nodes in the derivation graph that it can resolve with (given the search strategy and resolution refinement in effect) is found. The resulting *node pairs* are added to the *priority queue*, ordered according to a user-defined *cost function*.
3. Remove the "cheapest" node pair from the priority queue. If both nodes are active, apply the selected inference rule(s) to them. This will produce a set of *resolvents*, $\mathcal{R}$.[1]
4. If a *deletion strategy* is being used, then remove from $\mathcal{R}$ those resolvents that do not satisfy the corresponding condition(s). This can include, for example, forward subsumption and tautology, length and complexity deletion.
5. If a termination condition has been reached, stop. This is usually one of the following:
   - One or more *empty* or *answer* clauses have been found.
   - A maximum number of resolvents have been generated or integrated.
   - A time or memory limit has been reached.
   - No new resolvents can be produced.

   Otherwise, go to step 2. If *back subsumption* is used, at this point other nodes in the graph can be deactivated if their clauses are *subsumed* by newly integrated ones.

This is not quite the way things should *really* be done: A number of "tricks" can be used to improve the efficiency of the search, and the user is always free to design new ones. (Some of these are discussed in Sections 7, 8.1, and 12.)

[1] We use the term "resolvents," loosely, to refer to clauses obtained from *any* inference procedure, not necessarily resolution.

1.2 Overview

For an in-depth analysis of FRAPPS' design and the ideas behind it, see [Mitchell, 1989]. The purpose of this document is to describe the various functions provided to the user by FRAPPS, which have been classified according to the following categories:

- Modifying the derivation graph (Section 4)
- Accessing information stored in the derivation graph (Section 5)
- Inference rules (Section 6)
- Deletion strategies (Section 7)
- Search strategies (Section 8)
- FRAPPS Session Management (Section 9)

Before describing these categories in depth, Section 2 gives step by step instructions for getting FRAPPS "up and running." Section 3 describes the representation of clauses in the "language" of FRAPPS and the basic structures that are generated and maintained during a proof procedure. Section 10 provides a list of global variables, and Section 11 gives a simple example of how everything can be put together. Finally, Section 12 discusses various implementation issues that could be helpful to advanced users of FRAPPS.

The reader may notice that not all the functions described in this document are at the same level of functionality; that is, some of them can be defined easily in terms of others. This redundancy is included so that users can choose the functions already provided or define new ones, according to their own needs and preferences.

Throughout this manual, Lisp variable, constant, function and keyword names will appear in `modified typeface`, with arguments in *italics*. Square brackets (`[ ]`) will denote optional and keyword arguments. The character ">" will denote the Common Lisp top level system prompt.

2 Starting a FRAPPS Session

FRAPPS is initialized as follows:

1. Start up Common Lisp by typing `lisp` at the shell prompt.[2]
2. Load the initialization file into the Lisp environment by typing
   `(load ''frapps/init.lsp'')`
   at the Lisp top level system prompt.

[2] FRAPPS has been designed to run under any implementation of Common Lisp as described in [Steele, 1984]. However, a small number of global variables and path names might have to be changed depending on the particular installation.

3. Start FRAPPS by invoking the function `(start-frapps)`. This function loads all necessary files and initializes the system.

   If FRAPPS has been compiled, the user is given the option to use interpreted or compiled code. While the compiled code is obviously faster, the interpreted version can be useful during the process of developing and debugging functions written on top of FRAPPS.

   At this point FRAPPS should be ready to go.

3 Representing Clauses

3.1 The "Language" of FRAPPS

The "language" used to express clauses in FRAPPS is essentially a *list structured* version of First Order Predicate Calculus (FOPC), where all formulas are understood to be in clausal form. The various syntactic components of FOPC are represented as follows:

- **Constant, Function, and Predicate Symbols**

  Constant, function, and predicate symbols are denoted by Lisp atoms, excluding numbers and those specifically reserved by Lisp and FRAPPS. It is generally "safe" to use names consisting of combinations of letters and integers. A small number of symbols, however, are reserved by FRAPPS, including "`*var*`" and "`not`". Finally, note that Common Lisp does not distinguish between upper and lower case characters.

  Examples: `Jim`, `Garfield`, `one`, `c2`, `pet-of`, `barks`, `likes`.

- **Variables**

  Variables are denoted by lists of the form: `(*var*` *name subscript*`)`, where *name* is any symbol made up of letters and *subscript* is any positive integer. Two variables are equal if and only if they have the same name and the same subscript.

  As a convenience to the user, a variable macro is provided which allows variables to be specified by the shorthand notation: `?`*var-name*. This macro causes such a string to be expanded to the s-expression: `(*var*` *var-name* `1)`.[3]

  Examples: `?x`, `?thing`, which expand to `(*var* x 1)` and `(*var* thing 1)`.

- **Terms**

  A term can be a constant symbol, a variable, or a term made up of a function symbol and other terms. In the last case, a term has the following list structure:

  (*function-symbol term-1 term-2 … term-n*)

  where each *t*erm-i is an arbitrary term and *function-symbol* corresponds to an n-ary function, $n \geq 1$.

  Examples: `?x`, `(age Jim)`, `(weight (pet-of (*var* x 5)))`, `(sum ?x one)`.

[3]The FRAPPS function `(p-clause` *list*`)` can be used to "reverse" this conversion, printing the variables in *list* in a more readable form.

- `Atoms`

  Atoms are constructed from predicate symbols and terms, and have the following structure:

  (*predicate-symbol term-1 term-2 ... term-n*)

  where *term-i* is an arbitrary term as defined above and *predicate-symbol* corresponds to an n-ary predicate, $n \geq 0$.

  Examples: `(barks Jim)`, `(logician (friend-of ?x))`, `(likes ?x ?y)`, `(P)`, `(Q)`.

- **Literals**

  Literals are either atoms (*positive*) or the negation of atoms (*negative*). Positive literals have exactly the same structure as atoms; negative literals are obtained from positive literals by adding the symbol "`not`" as the first element of the list structure (i.e. *consing* it into the list).

  Examples: `(not barks ?x)`, `(likes ?x (friend-of ?x))`, `(P)`, `(not Q)`.

- **Clauses**

  Clauses are lists of literals, and thus have the following structure:

  (*literal-1 literal-2 ... literal-n*)

  where each *literal-i* is a literal and $n \geq 0$. If $n = 0$, then the clause is the *empty clause*, and is equal to `NIL`.

  Examples: `((barks ?x) (not barks (owner-of ?x)))`, `((P))`, `((not P) (Q))`.

- `Answer Predicates, Literals and Clauses`

  Any predicate beginning with the string "`answer`" is considered to be an *answer predicate*, and is treated accordingly. *Answer literals* are constructed from answer predicates, and are usually never negated.

  Examples: `(answer ?x ?y)`, `(answer-likes ?x garfield)`, `(answer-color ?x)`.

  Answer literals can be included anywhere in a clause. An *answer clause* is a clause, different from `NIL`, that contains only answer literals.[4]

  Examples: `((answer-one ?x) (answer-two ?y))`, `((answer jim))`

Note that it is the user's responsibility to ensure that function and predicate names are used consistently within the user-defined initial set of clauses. A predicate and a function should not be identified by the same symbol, and each predicate and function symbol should always be followed by the same number of terms (corresponding to its *arity*).

3.2 Frapps Node and Resolvent Structures

Frapps uses two basic data structures to represent clauses, depending on whether they have been integrated into the derivation graph or not.

[4]The Frapps function `(answer-lit-p` *list*`)` can be used to recognize answer literals.

3.2.1 Node-info structures

Nodes are created when a clause is integrated into the derivation graph, and are represented as `node-info` *structures*. A `node-info` structure contains the following fields:

- `id`
  An integer that *uniquely* identifies the node: Different nodes have different ids. (Note, however, that different nodes *can* contain the same clause.) FRAPPS assigns ids to nodes in ascending consecutive order, starting with 1, reflecting the order in which they are integrated into the derivation graph.

- `clause`
  The clause as represented in the "language" of FRAPPS.

- `parents`
  A list of the node-ids that identify the node's *parents*. If the clause is user-defined, this list is `NIL`. If factoring was used to derive the node, this list contains a single node-id. If binary or general resolution was used, the first element in the list is the node's *left parent*, and the second element is the node's *right parent*.

- `children`
  A list of the node-ids that identify the node's *children*.

- `level`
  An integer equal to the *derivation graph level* at which the node is found.

- `deriv-mthd`
  A symbol denoting the method used to derive the clause. The following symbols are the possible values of this field:

  - `UD`: the associated clause was *user-defined*
  - `BR`: the associated clause was derived by *binary resolution*
  - `GR`: the associated clause was derived by *general resolution*
  - `F`: the associated clause was derived by *factoring*

- `user-field`
  This field is reserved to be set at the user's convenience, and can contain any *Lisp object*. Its default setting is `NIL`.

- `active`
  A binary flag that is `NIL` if the given node has been *deactivated*, and `T` otherwise. Deactivated nodes do not participate in the priority queue and cannot be used to infer new ones (see Sections 6 and 8). Initially set to `T`.

Note that if s is a node-info structure, then the fields of s can be accessed as `(node-info-`*fieldname* s`)`, where *fieldname* is the name of a field as defined above.

3.2.2 Resolvent-info structures

A `resolvent-info` *structure* corresponds to a clause that has *not* been integrated into the derivation graph. Most of the inference, deletion and clause integration procedures take these as arguments and/or return them. They are basically `node-info` structures that lack `id`, `level`, `children` and `active` fields, and thus have the form:

- `clause`
- `parents`
- `deriv-mthd`
- `user-field`

These fields contain the same information as their counterparts in a `node-info` structure. In this case, if s is a `resolvent-info` structure, then the fields of s can be accessed with `(resolvent-info-`*fieldname* s`)`.

Both `resolvent-info` and `node-info` structures will be referred to simply as "clauses" if the context makes everything clear; thus one can speak of a clause's id, parents, level, and so on. (This could be misleading, since in general a clause can appear as two different nodes in the derivation tree.)

4 Modifying the Derivation Graph

The derivation graph can be modified in the following ways:

- A user-defined clause (a premise or an axiom) can be *introduced*, as a node, into the derivation graph.
- Previously inferred clauses can be *added*, as nodes, to the derivation graph.
- An existing node can be *temporarily suspended* from use in deriving new clauses.
- A currently suspended node can be *reactivated* so that it is once again eligible to participate in the derivation of new nodes.
- The derivation graph can be *cleared* of all currently existing nodes (that is, it can be "re-initialized"), with the option of retaining the (user-defined) base-set of nodes.

The following functions are used to carry out the actions listed above:

> `(def-clause` *clause* `[:user-field` *value*`] [:support-set t])`

This function integrates a node that contains the clause specified by *clause* into the derivation graph. This node is created at level 0, and its `deriv-mthd` field is set to `UD`. A unique integer *id* is automatically generated and assigned to the node. The node's *user field* can be initialized using the keyword argument [`:user-field` *value*].

If this argument is not specified, it defaults to `NIL`. Finally, if the keyword argument [`:support-set t`] is supplied, the clause is included in the *set-of-support*, which is indicated by the global variable `*support-set*`.

`def-clause` returns an updated list of the node-ids that have been defined up to, and including, the most recently defined node. This list is always available by referencing the global variable `*base-set*`. Typically, files are prepared defining in this way all the clauses to be used in a derivation. These files can then be loaded into the Lisp environment once FRAPPS has been set up. [5]

> (`integrate-clause` *resolvent-info-struct*)

Integrates a previously derived clause (i.e. a resolvent or factor) as a node in the derivation graph. The argument *resolvent-info-struct* must be a `resolvent-info` structure as described in Section 3.2. The function returns the *node-id* assigned to the new node.

All of the FRAPPS inference functions discussed in Section 6 return *lists* of resolvents in the form of `resolvent-info` structures.

> (`integrate-clause-list` *list-of-resolvent-info-struct*)

Performs the same function as `integrate-clause` except that instead of integrating a single derived clause as a node, it integrates a list of clauses as nodes, and returns the corresponding list of new *node-ids*. The argument *list-of-resolvent-info-struct* should be a list of `resolvent-info` structures.

All nodes are initially *active* when they are integrated into the derivation graph.

> (`deactivate-node` *node-id*)

Deactivates the node identified by *node-id*. This function provides a mechanism to *temporarily* suspend a node from being used to infer new clauses (i.e. the node cannot be used by `resolve` or `factor` to produce new clauses). Attempting to resolve upon or factor nodes which are currently deactivated will print a FRAPPS error message, and the inference operation will not be performed. (These messages can be turned off by setting the `*print-inference-errors*` global variable to `NIL`.) Note that the node remains in the graph, but *cannot* be used to derive new nodes. Information contained within deactivated nodes can, however, be accessed by the functions discussed in the next section.

> (`reactivate-node` *node-id*)

Reactivates the node identified by *node-id*, if previously deactivated. Upon being reactivated, the node can once again be used to derive new nodes via the inference functions described in Section 6.[6] The active state of a node is determined by the value of its `active` field.

[5] A simple program that converts FOPC formulas in OTTER syntax [McCune, 1990] into "FRAPPS-ready" clausal form is available; thanks go to William McCune for letting us use some of the OTTER code.

[6] Note, however, that by default a reactivated node is not considered in building the priority queue during the time it was inactive (see Section 8.1).

If the user should want to update the *user field* after assigning it an initial value, the following function can be used to *destructively* modify this field:

> `(set-user-field` *node-id value*`)`

Assigns to the field `usr-fld` of the node identified by *node-id* the value specified by *value*, which can be any *Lisp object*.

> `(reset-frapps)`

After a proof procedure has been completed, FRAPPS should be "reset" or "re-initialized" before a new one is run. The function `reset-frapps` re-initializes the derivation graph (i.e. *permanently* removes all nodes from the derivation graph) and resets all associated global variables. The *priority queue* (see Section 8.1) is also reset.

In some cases, the user will want to try a new search strategy on the same problem as before. In this case, the following function can be used:

> `(partial-reset)`

This function has the same effect as resetting FRAPPS and re-defining the base-set of clauses (using the current user-field and set-of-support values). The derivation graph is cleared of all nodes except for level 0, and the priority queue is "rebuilt" using the cost functions currently in place (see Section 8.1). All the database global variables are changed accordingly.

Note that the above two functions retain all flag and parameter values used in the previous derivation; that is, the defaults for all the global variables that do not depend on the derivation graph are *not* restored (see Section 10). These variables can be reset independently of the database using `(reset-defaults)`.

5 Accessing Information Stored in the Derivation Graph

Information is stored in the derivation graph in the form of *derivation nodes*, each a `node-info` structure. The derivation nodes are organized into levels, as follows: *User defined* nodes are inserted at level 0, and a derived node n is inserted at level `1 + max(`*levels of parents of n*`)`. Information can be accessed relative to a particular node, an entire level, or multiple levels of the derivation graph. Clauses are also indexed by length and the literals they contain, allowing for more complex access functions.[7]

5.1 Node Access Functions

For a particular derivation node, the information contained in the fields of the corresponding `node-info` structure can be accessed. This is done with a set of functions of the form `(get-node-`*fieldname cls-id*`)`, where *fieldname* is the name of the field in question as described in Section 3.2.

[7] WARNING: For efficiency reasons, most of the following functions that return complex Lisp objects (structures and lists) do *not* make copies of the structures in the corresponding databases. The user should never *destructively* modify any of the lists obtained in this manner.

That is, the following fields can be obtained using the corresponding functions:

- The clause associated with the node: `(get-clause` *node-id*`)`
- The node's parents: `(get-parents` *node-id*`)`
- The node's children: `(get-children` *node-id*`)`
- The level of the node in the derivation graph: `(get-level` *node-id*`)`
- The method used to derive the corresponding clause: `(get-deriv-mthd` *node-id*`)`
- The node's *active flag*: `(get-active` *node-id*`)`
- The user field associated with the node: `(get-user-field` *node-id*`)`

ALL of the above information concerning a particular node can be accessed using the function:

> `(get-node-info` *node-id*`)`

This function returns all of the above mentioned information concerning *node-id* in the form of a `node-info` structure (as described in Section 3.2). Note that a field containing *node-id* is also included; when entire levels of nodes are requested, ids can be used to distinguish one particular node from another.

All of the above functions make sure that the given *node-id* argument corresponds to a node in the derivation graph in order to avoid Lisp errors. If the given id does not correspond to any node in the graph, a FRAPPS error message is printed[8] and the functions return the atom `'F-ERROR`. While this feature is helpful during an interactive session, user-defined search procedures can usually avoid the extra overhead incurred in these checks. To this effect, a set of functions identified by `get-node-`*fieldname* is provided. These have the same behavior as their `get-`*fieldname* counterparts as described above, except that a Lisp error occurs if a node identified by *node-id* is not present in the derivation graph.

5.2 Accessing Nodes by Level

The levels in the derivation are indexed by consecutive integers, starting at 0 for the base-set of clauses. The global variable `*depth*` contains the greatest non-empty level in the derivation graph (and is initially set to 0).

> `(get-nodes-at-level` n`)`

This function returns a list of all derivation nodes that currently exist at level n of the derivation graph, in the form of `node-info` structures. For the user's convenience, an abbreviated name for this function is provided: `(gnal` n`)` is equivalent to `(get-nodes-at-level` n`)`.

Similarly, all nodes at all levels down to a particular level can be accessed by:

[8]The error messages can be turned off by setting the `*print-access-errors*` flag to `NIL`.

> `(get-nodes-downto-level [`*n*`] )`

This function (abbreviated to "`gndl`") returns a list of all nodes existing in the derivation graph at a level less than or equal to n, in the form of `node-info` structures. If the optional argument n is not specified, all of the nodes in the derivation graph are returned.

In the case that only a list of *node-ids* is desired, the following functions can be used:

> `(get-ids-at-level` *n*`)`

(abbreviated to "`gial`") returns a list containing the node-ids of all nodes at level n.

> `(get-ids-downto-level [`*n*`])`

(abbreviated to "`gidl`") returns a list containing the node-ids of all nodes at a level less than or equal to n. If the optional argument n is not specified, a list containing the node-ids of all the nodes in the derivation graph is returned.

Finally, the function

> `(print-level` *n*`)`

prints the clauses at level n, together with their parents and node-id. This function makes use of `print-clause`, so that variables are printed in a more readable form. Along with most other FRAPPS output procedures, it uses the `(user-print` *node-id*`)` function to print "extra" information about each node. Users can re-define this function at will (to print the contents of each node's *user field*, for example). By default, this function prints the value of the node's *active flag*.

5.3 Accessing Nodes by Clause Length

FRAPPS provides efficient access to nodes according to the length of their clauses. This is quite useful when implementing strategies such as *unit conflict*. The length of a clause is defined to be the number of literals in it, excluding *answer literals*.

The function `(clause-length` *clause*`)` is used by all FRAPPS procedures that need to determine the length of a clause. It returns the number of *non-answer* literals in *clause* (as opposed to Lisp's `(length` *clause*`)`, which would return the total number of literals).

> `(get-ids-of-length` *n* `)`

This function (abbreviated to "`giol`") takes an integer n as its argument and returns a list of the node-ids corresponding to all the clauses of length n in the derivation graph.

The global variable `*max-db-length*` contains the maximum length of a clause in the derivation graph.

5.4 Accessing Nodes by Literals

The most complex access functions available in FRAPPS reference a *clause literal database*, which is automatically maintained by FRAPPS as clauses are integrated into the derivation graph. These functions can be useful for implementing "advanced" features such as re-defining the priority queue node-pair creation process and pre-processing the base-set of clauses, and are now briefly described:

(`ret-clauses-cont-equal-lits` *literal*) returns the node-ids of all the clauses that contain a literal *equal* to *literal*.

(`ret-clauses-cont-unifiable-lits` *literal*) returns the node-ids of all the clauses that contain a literal *unifiable* with *literal*.

(`ret-clauses-cont-instance-lits` *literal*) returns the node-ids of all the clauses that contain *instances* of *literal*.

(`ret-clauses-cont-more-general-lits` *literal*) returns the node-ids of all the clauses that contain literals that are *more general* than *literal*.

5.5 Recovering Proofs

The global variables `*answer-cls-ids*` and `*empty-cls-ids*` list the ids of all the *answer* and *empty* clauses in the derivation graph.

For the user's convenience, the following functions are provided to help trace the derivation of any one particular node:

> (`get-derivation` *node-id*)

Returns a list corresponding to the derivation steps that generated (the node identified by) *node-id*. Each element of this list is of the form *((parent1-id [parent2-id]) node-id)*.

> (`print-derivation` *node-id*)

Prints the nodes used to derive *node-id*. Each clause is indented according to its level so that the general structure of the proof can be observed.[9]

6 Inference Rules

FRAPPS currently provides two inference rules, that can be used to derive new clauses from nodes in the derivation graph: *Binary resolution* and *factoring*. These two rules are combined to provide *general resolution*. In addition, the *SLD-resolution* refinement is also included in FRAPPS.

In general, the inference functions accept one or more node-ids as input. The corresponding inference rule is applied to the clause(s) associated with the given node id(s),

[9]Additional information for each clause can be printed by re-defining the (`user-print` *node-id*) function.

and the newly inferred clauses are returned as `resolvent-info` structures. Note that the results of an inference operation are NOT automatically added to the derivation graph; they must be *explicitly* added using the `integrate-clause` or `integrate-clause-list` functions described in Section 4.

The inference functions provided by FRAPPS correspond directly to the inference rules mentioned above, and are discussed in the following sections.

6.1 Binary Resolution

If the global flag `*factor-flag*` is `NIL`, the following function performs *binary* resolution:

> `(resolve` *node1-id node2-id* `[:c1-lit` *lit1#*`]` `[:c2-lit` *lit2#*`])`

Returns a list of resolvents of the clauses associated with *node1-id* and *node2-id*, where each resolvent is represented by a `resolvent-info` structure. If no resolvents are generated, `NIL` is returned.

Note that the user has the option of specifying the keyword arguments [`:c1-lit` *lit1#*] and [`:c2-lit` *lit2#*]. These arguments allow the user to specify a particular literal within each clause to resolve upon (i.e. to specify the selection, or "clash", literals). Both *lit1#* and *lit2#* should evaluate to integers corresponding to the position of a literal within the respective clause, where the first literal in the clause has index 1. The function `resolve` behaves differently depending on whether none, one, or both of the "clash" literals are specified.

1. **Neither of the optional arguments is supplied.**
   Given that neither clause has a specified selection literal to resolve upon, all literals from both clauses are treated as selected literals one at a time in a pairwise fashion. That is, `resolve` attempts to find *all* possible resolvents of the two clauses.
2. **One of the two optional arguments is supplied.**
   In this case one of the clauses has a specified selection literal, and the other clause does not. In this situation `resolve` forms all possible resolvents that result from resolving the specified selection literal against each of the literals in the other clause.
3. **Both optional arguments are supplied.**
   In the event that both selection literals are specified, `resolve` only attempts to form a resolvent from the two clauses by resolving upon the specified selection literals. Thus, at most one resolvent can result in this case.

In this case, the `deriv-mthd` flag of any resolvents is set to `BR`, to indicate that *binary* resolution was used to derive the corresponding clause.

For convenience, a specialized binary resolution procedure is also provided:

> `(binary-resolve` *node1-id node2-id* `[:c1-lit` *lit1#*`]` `[:c2-lit` *lit2#*`])`

This function behaves exactly as `resolve` does, except that binary resolution is always carried out, regardless of the value of `*factor-flag*`.

6.2 General Resolution

As mentioned earlier, FRAPPS supports both *binary* and *general* resolution. Whether general resolution is enabled or not is determined by the global variable `*factor-flag*`:

If the value of `*factor-flag*` is `NIL`, then `resolve` performs *binary* resolution. This is the default setting. A *non-nil* value specifies that `resolve` is to perform *general* resolution. In this case, `resolve` returns all the *general* resolvents of the two given clauses. The selection literal keyword arguments are *ignored* in this case, and the `deriv-mthd` flag of any resolvents is equal to `GR`.

In all of the above resolution procedures, the literals in each resolvent that originate in *node1-id* are placed before those from *node2-id*. Duplicate literals are then removed, from right to left.[10] The `parents` field of the new resolvents is always the list `(`*node1-id* *node2-id*`)`. (This is still the case when a clause is resolved against itself; the `parents` field of the resolvents will contain two identical ids.)

During an interactive FRAPPS session, the user might notice that the resolution functions return not only the `resolvent-info` structures corresponding to the resolvents, but also the *unifiers* used in obtaining them. This makes use of the *multiple-value return* feature of Common Lisp, and the list of unifiers can be accessed, if so desired, using the Lisp function `multiple-value-setq`.

One final note concerning which resolution procedure to use: From a performance standpoint, *binary* resolution will be much more efficient (since factoring is not performed), but suffers from being an incomplete inference rule by itself. The user should keep this fact in mind when choosing a "version" of `resolve`.

6.3 SLD Resolution

SLD resolution is a particularly important refinement of the resolution rule of inference, and the corresponding function is provided as a FRAPPS inference function:

> `(sld-resolve` *node1-id node2-id*`)`

returns a list containing the *SLD-resolvent* of the clauses corresponding to *node1-id* and *node2-id*, in the form of a resolvent-info structure. This resolvent is computed as follows:

One of the two clauses should be a *negative* clause (i.e. containing only negative literals), while the other should be a *definite* clause, containing exactly one positive literal. If these conditions are not met, `NIL` is returned. A *selection function* is used to determine which literal in the negative clause should *clash* against the one positive literal in the definite clause. If no such resolvent exists, `NIL` is returned.

Since the ordering of the literals in the resolvent may be important, we now describe how the SLD-resolvent is obtained: The "surviving" negative literals in the negative clause are appended to the negative literals in the definite clause, so that the latter go first (in PROLOG terms, the *new subgoals* go first). The literals are kept in the same relative order

[10] All the inference function remove duplicate literals. In this respect, FRAPPS treats clauses as *sets* of literals.

as in the original clauses. Duplicate literals are then deleted, starting from the right of the list and moving left.

Note that in SLD-resolution, binary resolution is always performed, so the resolvent's `deriv-mthd` field is always `BR`. The `parents` field of the new resolvent is equal to the list `(`*node1-id node2-id*`)`.

Users can define their own selection functions. The function used by `sld-resolve` is determined by the global variable `*sld-selection-fn*`, which should be set to the *name* of the function in question. In general, the selection function should take a *clause* as its single argument and return an integer that corresponds to the position of the selected literal within the clause. Literals are indexed within clauses starting at 1. The default selection function is `'leftmost-select-fn`, which returns the index number corresponding to the first non-answer literal in the negative clause (traversing the list in the usual left to right order). The converse function, `'rightmost-select-fn`, is also available.

6.4 Factoring

> `(factor` *node-id*`)`

Returns all *non-trivial* factors (i.e. factors which are *different* from the original clause associated with *node-id*) of the clause associated with *node-id.* If no such factors exist, `NIL` is returned; otherwise, all the non-trivial factors are returned in the form of a list of `resolvent-info` structures.

`factor` retains the basic ordering of the literals in the factored clause; for each set of "collapsed" literals, the leftmost one is retained (i.e. duplicates are removed from right to left in the instantiated literal list).

7 Deletion Strategies

Deletion strategies allow for the immediate elimination of derived clauses on the basis of some particular property that these clauses might possess. Without the use of deletion strategies, is it highly unlikely that a proof will be efficiently obtained, and in many cases no proof will be attainable at all within the limited computing resources at hand (assuming, of course, that there was a proof in the first place).

7.1 Tautology, Length and Complexity Deletion

The functions described in this section all accept a list of `resolvent-info` structures, *rslvnt-list*, as their input, and return the list of elements of *rslvnt-list* that are *not* deleted by the corresponding strategy. These functions are meant to be used *before* the `resolvent-info` structures are integrated into the derivation graph. Users can, of course, define their own deletion procedures, but the most common deletion strategies are already provided by FRAPPS:

Tautology deletion:

> `(delete-taut-resolvents` *rslvnt-list*`)`

Removes those resolvents in *rslvnt-list* whose clauses are *tautologies*. In general, the predicate `(taut-p` *clause*`)` returns `T` if *clause* is a tautology, and `NIL` otherwise.

Length deletion:

> `(delete-long-resolvents` *rslvnt-list*`)`

Removes from *rslvnt-list* those resolvents whose *clause length* is strictly greater than the global variable `*max-length*`. Note that clause length is computed ignoring answer literals.

Complexity deletion:

> `(delete-complex-resolvents` *rslvnt-list*`)`

Removes from *rslvnt-list* those resolvents whose *nesting depth* is strictly greater than the value of the global variable `*max-complexity*`. The nesting depth of a clause is defined as the maximum nesting depth of all its non-answer literals. The nesting depth of a literal is equal, in turn, to the maximum nesting depth, $\max\{D(t_i)\}$, over its arguments $\{t_i\}$ where $D(x) \stackrel{\text{def}}{=} 1$ if x is a variable or a constant, and
$D(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} \max(D(t_i), \ldots, D(t_n))$, for $n \geq 1$.

7.2 Back and Forward Subsumption

Subsumption tests can be both the most useful and the most expensive work carried out by a resolution-based theorem prover. Therefore, FRAPPS offers a variety of specialized subsumption procedures, which are chosen according to the values of a set of keyword arguments accepted by the FRAPPS functions that check for subsumption. As this test is weakened, the subsumption functions become more efficient, but "cover less ground." These arguments will be discussed after the functions have been introduced:

> `(delete-fwd-sub-resolvents` *rslvnt-list* `[:degree` *degree-flag*`] [:weak` *weak-flag*`] [:answer` *answer-flag*`] [:active` *active-flag*`])`

This function can be used to implement forward subsumption in a way similar to the deletion functions described in the previous section: It removes from *rslvnt-list* those resolvents which are *forward subsumed*, that is, whose clauses are subsumed by clauses already in the derivation graph. Note that the resolvents in *rslvnt-list* are *not* tested for subsumption amongst themselves.

The back subsumption function also takes a list of `resolvent-info` structures as an argument, but returns a list of the node-ids whose clauses are subsumed by the clauses in the list. These nodes are *deactivated* as a side effect:

> `(deactivate-back-sub-nodes` *rslvnt-list* `[:degree` *degree-flag*`] [:weak` *weak-flag*`] [:answer` *answer-flag*`])`

Deactivates those (active) nodes in the derivation graph whose clauses are *subsumed* by a resolvent in *rslvnt-list*. Deactivated nodes cannot be used to generate new resolvents or factors from then on (see Section 4). The list of node-ids corresponding to the subsumed clauses is returned. If the `*print-back-sub-clauses*` flag is set, then a message is printed for each deactivated clause.[11]

The degree to which subsumption is tested for in these two functions is set by the keyword arguments [`:degree` *degree-flag*], [`:weak` *weak-flag*] and [`:answer` *answer-flag*].

- The keyword argument [`:degree` *degree-flag*] can have the value `'full`, `'instances` or `'variants`. All subsumption procedures carry out the corresponding test, as follows:
  - **full**: In this case, FRAPPS detects "real" subsumption: A clause c_1 subsumes a clause c_2 if and only if there is a substitution θ such that $c_1\theta$ is a subset of c_2.
  - **instances**: FRAPPS only detects that c_1 subsumes c_2 if c_2 is an instance of c_1, that is, there is a substitution θ such that $c_1\theta = c_2$.
  - **variants**: FRAPPS only detects that c_1 subsumes c_2 if they are variants of each other, that is, there is a *renaming* substitution π such that $c_1\pi = c_2$. Note that this subsumption check can be used to ensure there are no duplicate clauses in the derivation graph.

  The default value for *degree-flag* is `'full`.

- If the keyword argument [`:weak` *weak-flag*] is non-nil, then FRAPPS only tests for subsumption if the subsuming clause is *shorter* than the subsumed clause. Default is `NIL`.

- If the keyword argument [`:answer` *answer-flag*] is `NIL`, FRAPPS ignores answer literals in all subsumption tests. That is, the tests are carried out as if the answer literals (if any) were not there. For example, the clause `((answer ?x) (red ?x))` would subsume the clause `(red a)`. If this variable has a non-nil value, then answer literals *are* taken into account. Default is `T`.

- In the case of forward subsumption, if the keyword argument [`:active` *active-flag*] is non-nil, only *active* nodes in the graph are considered as possible subsuming clauses. Default is `T`.

8 Setting up Search Strategies with FRAPPS

8.1 The Priority Queue

Search strategies can be implemented by defining a *cost function* on node pairs, under which they are ordered in a *priority queue*. Intuitively, the priority queue keeps track of *what to do next*: At each iteration of the main search loop, the "cheapest" node pair is popped from the queue. New clauses may then be derived from this pair, and are in turn integrated into the derivation graph. Finally, all the nodes in the derivation graph that each

[11] These messages can be modified by re-defining the function (`found-back-sub-msg` *id clause*), which takes as its arguments the subsumed node-id *id* and the subsuming clause *clause*.

newly integrated clause can resolve with are retrieved, and the corresponding node pairs are inserted into the priority queue.

In this way, the priority queue contains a record of all the possible resolution operations that have *not* yet been carried out. Node pairs are ordered on the basis of their "cost" as determined by the cost function in effect at the time they were added to the queue.[12]

Since new node pairs are automatically inserted into the priority queue at the time that a clause is integrated into the derivation graph, the priority queue is automatically maintained as the search proceeds. For the purpose of implementing derivation strategies, however, users must explicitly invoke the following function to retrieve the head element of the priority queue:

```
> (pop-priority-queue)
```

This function returns a pair of node-ids corresponding to the current *smallest* cost node pair in the derivation graph. It *destructively* modifies the priority queue, since the pair that is returned is no longer in the queue.

8.2 The Cost Function

The cost function is defined by a list of *cost component functions* ($f_1 \ldots f_m$) together with a list of *weights* ($w_1 \ldots w_m$). The *cost* of a given node pair $\{n_1, n_2\}$ is equal to

$$\sum_{i=1}^{i=m} f_i(n_1, n_2) \cdot w_i$$

where m is the number of cost function components in effect.

To re-define the cost function, users must assign a list of function names to the global variable `*cost-function-components*`. Additionally, the cost function weight vector `*cost-function-weights*` must be a list of as many weights as there are cost components. These weights can be either integer or real values.

In order to improve the efficiency of a search, users can specify a *maximum cost* that node pairs can have if they are to be part of the priority queue. This value is given by the global variable `*max-cost*`; node pairs whose cost function exceeds this value are not included in the queue. The default value of `*max-cost*` is `most-positive-fixnum` - 1.

A number of cost functions are predefined, including:

- **cls-length-costfn**: The cost is equal to the sum of the two node's *clause length* (does not count answer literals).
- **cls-depth-costfn**: The cost is equal to the maximum of the levels at which the two nodes reside in the derivation graph.
- **input-costfn**: Pairs where one of the clauses is a member of the base-set have a cost equal to the depth of their possible resolvents; all other node pairs have

[12] In some instances it is useful to think of the cost of a node pair as estimating the cost of all the potential resolvents of the two nodes.

`most-positive-fixnum` cost. (In this way, a breadth-first input resolution search is set.)

The default setting for `*cost-function-components*` is `'(cls-depth-costfn)`. The default setting for `*cost-function-weights*` is `'(1)`. These two defaults define a *breadth-first* search strategy.

Clearly, an infinite number of search strategies can be defined in this way. For instance, to set the cost of a node pair to be the sum of the length and depth of its nodes, the cost-function global variables are defined as follows:

```
>(setq *cost-function-components* '(cls-length-costfn cls-depth-costfn))
>(setq *cost-function-weights* '(1 1))
```

Users can write their own cost-function components. These should be of the form `(<user-function-name>` *node1 node2*`)`, where *node1* and *node2* are `node-info` structures. The function `<user-function-name>` should return a numerical value corresponding to the "cost" (whatever this means) of the node pair `(`*id1 id2*`)`. (The user can assume that the two given clauses will indeed resolve, since this should always be the case when the cost function is applied.)

8.3 Estimating the Cost Function

The attentive reader will note that the above implementation of some search strategies will, in general, be very inefficient. During the course of a search, the cost function will be applied to *all* possible pairs of resolvents, even in those cases where a better search scheme could rule out entire collections of them. This shortcoming can be corrected in part by using *single-node cost-function estimates*, as follows:

In addition to the *cost-function components*, FRAPPS can also use a cost function of a *single* node to build the queue. This function should (ideally) provide an underestimate of the cost of all resolvent pairs that include a given node. When this estimate is available, single nodes can be temporarily placed in the back of the queue if their cost is greater than that of already existing pairs. Single-node queue entries are expanded into pairs only when all "cheaper" pairs have been popped out of the queue, resulting in a much more efficient search. Furthermore, if the cost estimate of the single node exceeds the value of the `*max-single-cost*` global function, then the node pairs that would be generated when this node is placed into the queue are *never* considered.

The single-node cost function is defined by the global variables `*cost-function-single-components*` and `*cost-function-single-weights*`. The former should be a list of function names, each of which takes a single `node-info` *structure* as an argument, while the latter should be a list (of equal length) of numerical values. The single-node cost function is then defined in a way analogous to that of the clause-pair cost function.

The respective single-node "estimates" corresponding to the predefined cost-function components are:

1. `cls-length-single-costfn`
2. `cls-depth-single-costfn`
3. `input-single-costfn`

However, the default value for `*cost-function-single-components*` is a function that returns `most-negative-fixnum` on all values, and `*cost-function-single-weights*` is `'(1)`. This means that, as a default, no underestimates are used; if these two values are not changed, the priority queue will reflect exactly the ordering determined by the node-pair cost function.

Finally, the user can re-define the procedure that finds the nodes that should be paired up with single nodes to create new node pairs. See Section 12 for more details.

9 Frapps Session Management

During the course of a normal Frapps session, users will typically reach a point where they would like to stop, save the work they have done, and later resume the session from where they left off. Frapps provides a set of functions to do exactly this, by saving the current derivation graph and priority queue states in a Frapps *session file*.

In the following function descriptions, the *filename* argument should always be a string. Frapps session files are identified by the extension "`.ssn`", which is automatically appended to *filename*.

> `(save-session` *filename*`)`

Saves the *state* of the current Frapps session in the file *filename*`.ssn`. The state is composed of the set of Frapps global variables as described in Section 10, all the nodes in the derivation graph, and the current state of the priority queue. If the file *filename*`.ssn` already exists, a Frapps error message is printed and the save is *not* carried out.

> `(resume-session` *filename*`)`

Restores the Frapps session contained in the file *filename*`.ssn`. If this file is not found, a Frapps error message is printed and the previous Frapps state remains unchanged.

> `(delete-session` *filename*`)`

Deletes the *previously saved* Frapps session stored as *filename* (that is, the file *filename*`.ssn` is *removed*). If the specified file is not found, the user is notified of this fact.

Note that the two preceding functions are restricted to manipulate only Frapps session files. A Frapps message is printed if this is not the case, and no files are created or changed.

Finally, if the user needs to save only the state of the derivation *graph* for later inspection (for example, after a proof has been found), the function

> `(save-derivation` *filename*`)`

can be used. This function saves the derivation graph and FRAPPS global variables, but not the priority queue: A considerable amount of space can sometimes be saved by not including it. A derivation saved in this way can be read back in with `(resume-session` *filename*`)` as usual, but the priority queue will always be empty in this case.

9.1 Saving User Fields and other Global Variables

By default, FRAPPS assumes that user-fields, if used, are atoms or lists. If this is not the case, then the user should define a pair of functions that translate user fields into lists and back:

`(user-field-to-list` *user-field*`)` should return a list "equivalent" to *user-field*.
`(list-to-user-field` *list*`)` should return a user-field "equivalent" to *list*.

By default, these two functions are the identity, meaning that FRAPPS assumes that user fields are simple *s-expressions*.

Finally, the user can save a fixed set of non-FRAPPS variables within a session file by defining the following two functions:

`(list-user-globals)` should return a list of values (atoms, numbers or lists) to be saved.
`(restore-user-globals` *list*`)` should restore whatever global variables were previously saved as *list* using `list-user-globals`.

10 Global Variables

A number of FRAPPS global variables that can be of interest to the user are listed and discussed briefly in this section. These variables have been divided into three groups.

The function `(print-globals)` can be used to produce a listing of the current values of all FRAPPS global variables.

10.1 Database Global Variables

The first group of variables are those that convey information about the derivation graph, and are automatically maintained by FRAPPS.[13] The function `(print-graph-globals)` displays the values of these variables:

`*base-set*`

A list of the node ids for the input set of clauses. These are the user-defined nodes, and are all at level 0.

[13] WARNING: The user is responsible for the consequences of changing the value of any of these variables. In particular, the values of `*last-id*`, `*depth*` and `*max-db-length*` should *never* be changed.

`*support-set*`
: A list of the node ids for the input clauses defined as belonging to the *set-of-support*.

`*last-id*`
: Contains the (integer) identifier of the most recently integrated node.

`*depth*`
: Contains the largest depth of any node in the derivation graph (i.e. the largest non-empty level in the graph).

`*max-db-length*`
: Maximum clause length over all nodes in the derivation graph.

`*empty-cls-ids*`
: Specifies a list containing the node identifier(s) of the empty clause in the event that it is integrated into the derivation graph. Note that the empty clause can be generated and integrated more than once, depending on how the search is conducted.

`*answer-cls-ids*`
: The counterpart to `*empty-cls-ids*`, this variable keeps track of the *answer clauses* integrated into the derivation graph. An answer clause is a non-nil clause that contains only answer literals.

10.2 Flags and Parameters

All the other global variables in FRAPPS can be set at the user's choice. Their values will usually determine the way a derivation is conducted within FRAPPS. The function `(print-flags)` prints their values, which can be restored to the default ones with the `(reset-defaults)` function.

The global variables that define the priority queue are described separately in Section 10.3, while the rest now follow:

`*factor-flag*`
: Determines whether factoring is carried out as used by the `resolve` function, enabling/disabling general resolution. Default: `NIL`, meaning that factoring is disabled. (See Section 6.)

`*sld-selection-fn*`
: Selection function for the SLD-resolution refinement (see Section 6).
Default: `'leftmost-select-fn`.

`*max-length*`
: An integer that specifies the maximum length (i.e. number of *non-answer* literals) that a clause can contain without being deleted if the length deletion strategy is being used (see Section 7). Default: `most-positive-fixnum`.

`*max-complexity*`
: An integer that specifies the *maximum* allowed nesting depth of a clause if complexity deletion strategy is being used (see Section 7). Default: `most-positive-fixnum`.

`*print-back-sub-clauses*`
: Indicates whether `deactivate-back-sub-nodes` prints a message for each back subsumed node (see Section 7).

`*print-access-errors*`
: Determines whether error messages are printed when the `get-node-`*fieldname* access functions are given invalid node-ids. (see Section 5)

`*print-inference-errors*`
: Determines whether error messages are printed if an inference rule is applied to deactivated nodes. Default: `T`. In particular, users might want to set this variable to `NIL` when running an automated proof procedure that uses back subsumption or otherwise deactivates nodes (see Section 7).

10.3 Priority Queue Global Variables

The following are the parameters that determine the behavior of the priority queue:

`*queue-on*`
: Specifies whether the priority queue mechanism is being maintained.

  In some cases, the user might want to write search procedures that do *not* use the priority queue. The global variable `*queue-on*` determines whether newly integrated nodes are added to the queue. The additional overhead and memory involved in maintaining the queue can be avoided by disabling it, by setting this variable to `NIL`. (To preserve the possible completeness of a search, it is strongly recommended that this variable remain constant during the entire course of a search.) (see Section 8). Default is `T`.

`*pair-select-function*`
: The name of the function used to generate the node pairs that are integrated into the priority queue at the time that a single node is expanded into a set of node pairs. In general, a pair selection function `(function` *node-id*`)` is expected take a node-id as an argument and return a list of node-ids that will be paired up with *node-id*. Default: `'gen-pair-select-fn`. (See Section 12 for more on this function.)

The priority queue cost function is defined by the following global variables:

`*cost-function-components*`
: Default: `'(cls-depth-costfn)`.

`*cost-function-weights*`
: Default: `'(1)`.

`*max-cost*`
: Default: `most-positive-fixnum` - 1

`*cost-function-single-components*`
: Default: `'(most-negative-costfn)` (returns `most-negative-fixnum` on all values).

`*cost-function-single-weights*`
Default: `'(1)`.

`*max-single-cost*`
Default: `most-positive-fixnum` - 1

Note that the weight vectors should have exactly as many elements as the corresponding component list.

11 Putting it all Together

To illustrate the simplicity of constructing resolution refutation systems using FRAPPS, an example is provided in this section. User-defined functions will typically embody much of the same structure that is illustrated here. One or more inference operations are relied upon to generate resolvents, are then judged according to some pre-defined criterion. Those judged as "good" are integrated, and the process is repeated until some termination condition is reached (in this case, until an empty or answer clause is found, or the priority queue is empty).

The function `user-search` provides a basis from which many different strategies can be constructed. Note that this function relies on the FRAPPS function `integrate-clause-list` in order to construct the derivation graph. The function `user-resolve` determines which inference strategy is used. The function `user-delete` is a combination of built-in FRAPPS deletion strategies.

We first use the default breadth-first search strategy to generate a proof of the *empty clause* using unrestricted binary resolution, with restrictions on the length and complexity of integrated clauses:

```
(defun user-search ()
  (setf clause-pair (pop-priority-queue))
  (do () ((or (not clause-pair) *empty-cls-ids* *answer-cls-ids*))
      (integrate-clause-list
       (user-delete
        (user-resolve (first clause-pair) (second clause-pair))
        ))
      (format t "~d " *last-id*) ;; Prints id of last integrated node.
      (setf clause-pair (pop-priority-queue))
      )
  (if *empty-cls-ids*
      (format t "~%Found empty clause, node ~d" *empty-cls-ids*))
  (if *answer-cls-ids*
      (format t "~%Found answer at node ~d, ~d" *answer-cls-ids*
              (get-clause *last-id*))))

(defun user-resolve (id1 id2) (binary-resolve id1 id2))
;; Choose binary resolution.
```

```
(defun user-delete (resolvent-list)
  (delete-taut-resolvents                ;; Choose tautology deletion
     (delete-complex-resolvents             ;; Choose complexity deletion
        (delete-long-resolvents resolvent-list) ;; Choose length deletion
        )))
```

(Note that the most expensive test in `user-delete`, tautology deletion, is used as the outermost one, while the least expensive, length deletion, is the innermost.)

To make the strategy work on a predefined set of base clauses:

```
(reset-frapps)
(load "filename.lsp")
;; This file should contain the definition of a base-set of clauses

(setq *max-length* 5) ;; Specify maximum clause-length allowed
(setq *max-complexity* 5) ;; Specify maximum nesting depth

(user-search) ;; This should print the last integrated node-id for
              ;; each pair of nodes that is resolved. By default,
              ;; the search is carried out in a breadth-first manner.
```

To carry out, say, unit *general* resolution, weighing single clauses according to the sum of their depth and length, and using only tautology deletion, we would now do:

```
(setq *cost-function-components* '(unit-costfn))
(setq *cost-function-weights* '(1))
(setq *cost-function-single-components*
  '(cls-length-single-costfn cls-depth-single-costfn))
(setq *cost-function-single-weights* '(1 1))

(defun user-resolve (id1 id2) (resolve id1 id2))
(defun user-delete (resolvent-list) (delete-taut-resolvents resolvent-list))

;; The system has to be reset and the base-set of clauses loaded again.
;; (If the problem set is the same, then "(partial-reset)" can be used.)
;; Note that we do this AFTER the cost function is redefined.

(reset-frapps)
(load "filename.lsp")

;; Binary resolution is the default (*factor-flag* is NIL),
;; so we have to do this:

(setq *factor-flag* T)

(user-search)
```

Base-clause set examples can be found in the "`frapps-demos`" directory under filenames of the form "*name*`.lsp`." These demos can be loaded with the function `(load-demo ''`*name*`'')`. The function `(run-demos)` can be used to attempt to solve a number of predefined sample problems using simple search strategies.

12 Advanced Frapps Programming - Implementation Tips

The following are various odds and ends that may help the user design more efficient search procedures using Frapps:

12.1 Subsumption Flag and Degree Settings

- Subsumption checks are, in general, *very* expensive. A correct setting of the keyword arguments in the back and forward subsumption procedures may reduce this cost while still maintaining a worthwhile deletion strategy.

  In the current Frapps implementation, the `:answer` keyword argument should be left at its default value of **T** if answer extraction is not being done. However, if answer extraction *is* carried out, the best setting for the `:answer` keyword argument is not obvious. In general, this flag's `T` setting is useful to carry out subsumption deletion yet retain all *different* answer clauses in the search space. It should probably be set to `NIL` unless multiple answers are being searched for.

  In any case, the setting of the subsumption flags pose complex trade-off considerations; the nature of the search space determines what values will be more efficient.

- The `'variants` setting for the `degree` flag provides a relatively inexpensive way for deleting and/or deactivating duplicate clauses in the derivation graph.

  If `'full` subsumption is used, users might want to choose to carry out back subsumption only when small clauses are integrated. Clauses with few literals will usually back subsume a larger number of nodes, and testing for this subsumption will be considerably less expensive than in the case of longer clauses.

12.2 Interactive Subsumption Detection Functions

The following procedures can be useful to collect information about the derivation tree, do subsumption checks, or both, during an interactive session with Frapps. They can also be useful in *pre-processing* the initial set of clauses, since they provide a mechanism for checking subsumption amongst them.

The function `(get-ids-sub-by-clause-list` *clause-list*`)` takes as its argument a list of clauses and returns the ids of all the nodes subsumed by them;
`(get-ids-sub-by-clause` *clause*`)` does the same for a single clause. (These functions are abbreviated to `giscl` and `gisc`, respectively.)

On the other hand, `(get-ids-that-subsume-clause` *cls*`)` takes as its argument a clause and returns a list of all the node-ids whose clauses subsumes it. (Abbreviated to `gitsc`.)

The function (`get-ids-sub-by-node` *node-id*) returns a list of all the node-ids whose clauses are subsumed by the clause identified by *node-id*, except for *node-id* itself. (`get-ids-sub-by-node-list` *node-id-list*) does the same for a set of node-ids. (Abbreviated to `gisn` and `gisnl`.)

Finally, the function (`get-ids-that-subsume-node` *node-id*) returns the node-ids whose clauses subsume the clause identified by *node-id*. (Abbreviated to `gisn`).

All of the above functions accept the keyword arguments `:degree`, `:answer` and `:weak`, which play the same role and have the same default values as those for the subsumption procedures in Section 7.2. Like the forward subsumption procedure, they also accept a fourth keyword argument, [`:active` *active-flag*]. If this flag is `NIL`, then *both* active and inactive node-ids in the derivation graph are considered. Again, this argument defaults to `T`; that is, deactivated node-ids are *not* considered or returned.

12.3 Search Tricks

- **Unit Conflict**

  A simple procedure known as *unit conflict* can increase the effectiveness of most search procedures. Whenever a *unit clause* (a clause of length 1) is generated, all other unit clauses in the derivation graph are examined in the hopes of finding a clash, which would immediately produce a proof. Such a procedure can be easily implemented in FRAPPS using the length and literal access functions described in Section 5.

- **Set-of-support**

  To "hard-wire" the set-of-support strategy into a search procedure, it suffices to make sure that no two *unsupported* base-set clauses (that is, user-defined clauses not in the set-of-support) are ever resolved. The properties of the priority queue can be used to enforce this at almost no cost: Simply set the single-node cost function to be "infinite" for unsupported base-set clauses during the time that the base-set is defined. In this way, the priority queue will initially contain only base-set clauses in the set-of-support. Assuming that no new user-defined clauses are added during the search, the cost function can then be reset to define the actual search strategy, without ever worrying again about the set-of-support. (Note that the intended ordering should also be used to order the base-set nodes in the set-of-support.)

- **A single-node priority queue**

  The priority queue can be used, in effect, as a stack of single nodes, by setting the *node-pair* components to zero. In this way, the only actual node *pairs* in the queue are those corresponding to the "cheapest" *single* node in the queue. This is particularly useful when implementing simple strategies such as breadth-first search. In conjunction with set-of-support and unit conflict, this approach yields the *given-clause algorithm* used by OTTER [McCune, 1990].

- **Modifying the node-pair creation process**

  When a single node is popped from the priority queue, FRAPPS retrieves all the nodes in the graph that it can resolve with, and adds them to the priority queue. It is *then* that some of the node pairs get screened out, on the basis of their cost function value.

This node-pair retrieval process can be re-defined by the user, by setting the global variable `*pair-select-function*` to the name of a function of the following form:

`(my-pair-select` *id*`)` should return a list of ids, corresponding to those nodes that can be "matched" with the node *id*. The corresponding pairs are then added to the queue.

The default value is `gen-pair-select-fn`; this function returns `NIL` if the node is inactive, or else the ids of all the active nodes in the graph containing a literal that resolves with a literal in *node-id*'s clause; furthermore, to avoid duplication of work, this function always returns only node-ids smaller than or equal to *node-id*. In this way, node pairs are treated as sets, and none is ever popped twice from the queue.

A specialized node-pair selection procedure for SLD-resolution is also available: `(sld-pair-select-fn` *node-id*`)` returns `NIL` if *node-id* corresponds to a deactivated or a non-negative clause in the derivation graph. Otherwise, it returns the ids of all active clauses that contain a literal that clashes with the literal in the node id's clause, as selected by the current SLD-selection function (given by the global variable `*sld-select-fn*`). Note that ordering pairs by node-ids is not necessary in this case.

12.4 Integrating Constraint-solving Unification

FRAPPS has been extended to support the introduction of an arbitrary unification procedure, which can take into account constraints associated with each clause. This feature can be used to construct *hybrid reasoners* in accordance to the *substitutional framework* [Frisch, 1991]. The extended system can be used, for example, to incorporate specialized equational reasoning through E-unification, or taxonomic reasoning by using sorted logic and sorted unification.

This extension is documented in an appendix to this manual [Uribe, 1991]. The user is responsible for implementing unification of pairs of terms, for use in binary resolution, and, optionally, unification for sets of terms, for use in factoring and general resolution. The new unifier can produce an arbitrary finite number of unifiers; each should be identified by a substitution and a new set of constraints. If back and forward subsumption are to be used, the user should also provide functions that carry out subsumption tests under the new constraints: In general, there is no procedure for obtaining a subsumption algorithm given only an arbitrary unification procedure.

References

[Frisch, 1991] Alan M. Frisch. The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. *Artificial Intelligence*, 49:161–198, 1991.

[McCune, 1990] William W. McCune. OTTER 2.0 users' guide. Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, March 1990.

[Mitchell, 1989] Michael K. Mitchell. The anatomy of a framework for resolution-based automated proof procedure systems. Master's thesis, University of Illinois, Urbana, Illinois, July 1989.

[Steele, 1984] Guy L. Jr. Steele. *Common Lisp: The Language.* Digital Press, 1984.

[Uribe, 1991] Tomás E. Uribe. *Constraints in* FRAPPS *2.0.* University of Illinois, Urbana, Illinois, September 1991.