CONSTRAINTS IN FRAPPS 2.0
USERS' GUIDE

Tomás E. Uribe

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

September 28, 1991

1 Introduction

FRAPPS (the Framework for Resolution-based Automated Proof Procedures) Version 2.0 has been extended to support the introduction of specialized unifiers. This feature can be used to construct *hybrid reasoners* in the spirit of the *substitutional framework* as described in [Frisch, 1991].

This document presents the resulting system, called *hooked-on*-FRAPPS (or H-FRAPPS, for short), and is intended to serve as an appendix to the FRAPPS 2.0 Users' Manual ([Mitchell *et al.*, 1991]). Users are referred to the FRAPPS manual for a description of the basic functionality of the system, and to [Mitchell, 1989] for an in-depth description of the original design of FRAPPS.

H-FRAPPS retains all of the features provided by FRAPPS, only now parameterized by a user-defined unification procedure. This unifier should take into account, and generate, *constraints*, that are associated with each clause, by returning substitutions that map constrained variables into constrained terms.

This extension has been designed to make the process of integrating a constraint unification algorithm into FRAPPS easier. However, this does require understanding some of the internal workings of FRAPPS, which this manual also describes. Unless otherwise noted, all features, functions and global variables described in the FRAPPS 2.0 manual are also included in H-FRAPPS.

1.1 Overview

Following the organization of the FRAPPS manual, Section 2 describes how H-FRAPPS is initialized. Section 3 describes the extended representation of clauses in H-FRAPPS, including *constraint* fields and a slightly more flexible syntax for variables. Section 4 describes the format required of the user-defined unification procedure. Section 5 describes resolution and factoring using the new unifier, and Section 6 discusses subsumption under the new constraints.

Section 7 describes changes in the *session management* functions. Section 8 lists the new global variables, as well as those functions that may have to be redefined. Finally, Section 9 describes, as an example, a simple sorted unification procedure that is included together with H-FRAPPS.

2 Starting up H-FRAPPS

H-FRAPPS is initialized in a manner similar to that of FRAPPS, as follows:

- Instead of loading the "`init.lsp`" file, users should load the "`hinit.lsp`" file.[1]
- As before, the function `(start-frapps)` loads all the necessary files. The user might notice that some of these files are also part of FRAPPS; this is because H-FRAPPS is simply an *extension* of FRAPPS, rather than a new version of it. As usual, users can choose between compiled and interpreted code, if the system has been compiled.[2]
- If the user has developed a new unifier, the corresponding files should be loaded at this point. (The default unifier is that for unconstrained terms; in this case, H-FRAPPS behaves exactly like FRAPPS – see Section 4.) Files that support sorted unification under monomorphic sort theories can be loaded with the function `(load-simple-sorts)` (see Section 9).

3 The Constraint Language

The syntax of clauses in H-FRAPPS is basically the same as that of FRAPPS. The only differences are an extended definition of variables and a new constraint field associated with both *node-info-structures* and *resolvent-info-structures*:

3.1 Variables

Variables in H-FRAPPS are lists of the following form:

`(*var*` *name subscript* ...`)`

The first element of the list is always `*var*`; *name* is any atom, and *subscript* is a positive integer. These three elements can then be followed by an arbitrary number of arbitrary s-expressions: This is allowed so the user can include extra constraint information in the variables themselves.[3] The rest of the list can also be `NIL`; in this case, variables are syntactically identical to those in FRAPPS.

[1] The particular pathname leading to this file will, of course, change depending on the installation.

[2] Note: Compiled files cannot, in general, be shared between FRAPPS and H-FRAPPS.

[3] This is particularly useful in the case of simple sorted logic; the fourth element of the list is the variable's *sort* (see Section 9).

3.2 Constrained Clauses

H-FRAPPS provides two different ways to associate a constraint with a clause. Constraints can be made part of the variables themselves, as shown in the previous section, or can simply be described by the new `constraints` field of node-info and resolvent-info structures. The first option is recommended if constraints are *unary*; that is, if the constraints on each variable are independent of those on all other variables. In this case, the constraints themselves should *never* contain variables.

The second option is more general and is provided for implementing complex constraints. The new `constraints` field can contain any Lisp object, although atoms and lists are recommended. (If some other type of Lisp object is used, the user will have to redefine some functions that depend on the internal structure of constraints – see Sections 5.1 and 7).

Node-info structures represent clauses that have been integrated into the derivation graph. If *struct* is a node-info structure, then (`node-info-constraints` *struct*) will return its constraint field. The following two functions can be used to access and modify the constraint field of a given node:

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

This function returns the constraint field of the node identified with *node-id*. If such a node does not exist, an H-FRAPPS error message is printed and the function returns `F-ERROR`.[4]

> (`set-constraints` *node-id value*)

This function can be used to modify the constraint field of a given node:
It sets the `constraints` field of the node denoted by *node-id* to *value*, *destructively* modifying this field.

Resolvent-info structures are used to represent clauses not yet integrated into the derivation graph, and now also contain a `constraints` field. If *struct* is a resolvent-info structure, then (`resolvent-info-constraints` *struct*) can be used to access its constraint field.

3.3 Defining Constrained Clauses

The `def-clause` procedure now accepts a third optional keyword argument, corresponding to the user-defined clause's constraint field, and thus has the following form:

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

If the [`:const` *value*] keyword argument is specified, then the `constraints` field of the corresponding node-info structure is initialized to *value*. Otherwise, this field is set to `NIL`.

[4] In keeping with the access functions provided by FRAPPS, the function (`get-node-constraints` *node-id*) will do the same without conducting error checking.

4 Defining a New Unifier

The new, user-defined unifier lies at the core of this extension of FRAPPS. It is particularly crucial, then, that the arguments and return values of this function be defined in accordance to what H-FRAPPS expects.

4.1 Binding Lists

Binding lists describe substitutions, which are mappings from variables to arbitrary terms. A binding list in H-FRAPPS is an s-expression with the following structure:

`(list (cons` *var-1 term-1*`) (cons` *var-2 term-2*`)` ... `(cons` *var-N term-N*`) )`

were *var-i* is a variable (as defined in Section 3.1) and *term-i* is an arbitrary *s-expression* (possibly containing variables); *var-i* and *var-j* should be different for $i \neq j$. The identity substitution is `NIL`.

Note that the components of the binding list (the *bindings*) are obtained by *consing* the variable **directly** into the term; this is necessary so that binding lists can be used as Lisp *association lists*. For example, the substitution mapping `(*var* x 1)` into `(*var* y 1)` and `(*var* z 1)` into the constant `a` would be represented as:

`(((*var* x 1) *var* y 1) ((*var* x 1) .  a))` [5]

4.2 Arguments

The user-defined unifier for H-FRAPPS should be a function of the following form:

> `(my-unifier` *exp1 exp2 const1 const2*`)`

The first two arguments, *exp1* and *exp2*, should be allowed to be arbitrary s-expressions, possibly containing variables in the format of H-FRAPPS (see Section 3.1). The arguments *const1* and *const2* are the constraints associated with the two expressions to be unified.

The global variable `*unifier*` should be set to the name of the new unification procedure. (Its default value is `'simple-unify`, a function defined in the following section.)

4.3 Return Value

The value returned by the user-defined unification procedure should be such that H-FRAPPS can *apply* the corresponding unifiers without having to know anything about the internal structure of constraints. Therefore, it is expected to have the following form:

`((`*binding-list-1 const-1*`) (`*binding-list-2 const-2*`)` ... `(`*binding-list-N const-N*`) )`

[5] Recall that "." denotes a Lisp *cons* structure.

where each *binding-list-i* is a binding list as defined in Section 3.1 and *const-i* is a constraint corresponding to this particular unifier. If the unification fails, the function should return the atom `'FAIL`.

The following points are important to keep in mind when designing the new unifier:

- If the unification succeeds, the return value should be a *list* of pairs, each of which is a solution to the given unification problem. This should be the case even when only one unifier is produced (since H-FRAPPS has no way of predicting the unifier's behavior).
- Each unifier (*binding-list-i const-i*) will be "applied" to a constrained clause by applying the substitution *binding-list-i* to the clause itself and simply replacing the original *constraints* field by the new *const-i*.

  Therefore, each returned constraint *const-i* should contain whatever information present in *const1* and *const2* that is still applicable to terms obtained from applying the substitution defined by *binding-list-i*.

As an example of the input-output behavior expected of the unifier, here's how the unconstrained unification function, `unify`, is adapted to H-FRAPPS. (The FRAPPS function (`unify` *exp1 exp2*) returns the most general substitution – a binding list – that unifies the two s-expressions *exp1* and *exp2* if the two expressions are unifiable, and `FAIL` otherwise.)

```
(defun simple-unify (lit1 lit2 &optional const1 const2)
  (declare (ignore const1) (ignore const2))  ;; For the compiler
  (let ((result (unify lit1 lit2)))
       (if (not (eq result 'FAIL))
           (list (list result nil))
           'FAIL)))
```

If the arguments *lit1* and *lit2* unify, this function returns a *list* containing a single pair: The binding list for the (regular) unifier and the `NIL` constraint. Otherwise, `FAIL` is returned. The arguments *const1* and *const2*, if present, are ignored.

4.4 The Factoring Unifier

The implementation of factoring and general resolution becomes significantly more complex if multiple unifiers can be produced by the new unification algorithm. A number of H-FRAPPS global variables can be set to make this procedure more efficient.

The global variable `*singleton*` should indicate whether the function identified by `*unifier*` can return multiple unifiers. To improve efficiency, `*singleton*` should be set to a non-nil value if at most one unifier can be produced. Otherwise, it should always be set to `NIL`, which is its default value.

If multiple unifiers can be produced, defining a second unification procedure is *strongly recommended* in order to carry out more efficient factoring. The return value expected from this function is a set of unifiers in the same format as above; but now the arguments are as follows:

> `(my-factoring-unifier` *list const*`)`

where *list* is a set of terms and *const* is a constraint field. This procedure should return a list of unifiers for the set of expressions contained in *list*, which are *all* constrained by *const*. If the unification fails, the atom `'FAIL` should be returned. (The literals factored upon will make up the set of expressions, and the given constraint will be that of the factored clause.)

In this case, the global variable `*general-unifier*` should be set to the name of this procedure. By default, H-FRAPPS will use the unification procedure determined by the global variable ***unifier*** to obtain factors (see Section 5.2), using a predefined function called `general-unify`: This is the default value of `*general-unifier*`.

Note: The specialized factoring procedure is *ignored* if the global variable `*singleton*` is not `NIL`.

5 Inference Procedures

Like FRAPPS, H-FRAPPS provides binary resolution and factoring, combining them to obtain general resolution. In H-FRAPPS, the new unification procedure is used to unify the appropriate sets of literals under the corresponding constraints.

5.1 Renaming Variables

When resolving two clauses, the variables in one of them are automatically renamed by H-FRAPPS before the actual resolution is performed. This poses an implementation problem, since the constraints associated with the clause also have to be modified to reflect this change. Users should be aware of how this process is carried out so that they can modify the constraint structures accordingly:

H-FRAPPS renames variables by incrementing their subscript by an appropriate amount. It is assumed that all the variables in FRAPPS format modify (i.e. lists beginning with `*var*`) contained in the `constraint` field of a node-info structure are in the same "scope" as all the other variables in that field and the variables in the corresponding clause. This way H-FRAPPS can go through the constraint and rename all variables in the same manner as the variables in the clause were renamed, without affecting the semantics of the clause/constraint pair.

If the constraint field is *not* a list, then the user should redefine the following function:

> `(increment-constraint-subscripts` *const increment*`)`

This function should return the constraint structure obtained by incrementing all the variable subscripts in the constraint *const* by the amount *increment*.

In doing this, the following FRAPPS function should be useful:

> `(inc-var-subscrs` *list increment*`)`

returns the result of adding *increment* to the subscripts of all the variables appearing in the (possibly nested) list *list*.

The pre-defined version of `increment-constraint-subscripts` assumes that the constraint *const* is a list, and thus is equivalent to `inc-var-subscripts`.

5.2 Factoring

As mentioned in Section 4.4, if more than one most general unifier can be returned by the new unification procedure then the efficiency of the factoring procedure can usually be improved. This relies on a second user-defined unification procedure, which finds a minimal set of unifiers for a *set* of terms (see Section 4.4). H-FRAPPS can then apply this function to unify the sets of literals that are factored upon.

In the event that this function is not available, FRAPPS unifies a set of literals using the function determined by `*unifier*`, (which unifies only *pairs* of terms). This can easily lead to an unnecessarily large number of unifiers, most of which are less general than others.

Note that this affects the efficiency of not only factoring itself, but also general resolution, in which factoring is a basic step. However, it is not an issue if only binary resolution is used.

5.3 Resolution

In a binary resolution step, the constraints of the two resolvents are given to the unifier together with the literals to be clashed. The new clause is formed by applying the given substitution to the literals that form the resolvent, and adding the new constraint.

In the case of general resolution, all factors of the two clauses are first obtained, and then all possible binary resolvents of between the new factors under their new constraints are generated.

6 Subsumption

There is no automatic procedure for detecting subsumption given only a unification algorithm.[6] Therefore, if users want to use the back or forward subsumption deletion procedures, they will have to *define* the following functions (with the same names as they appear here):

[6] In fact, there are constraint languages for which the problem of subsumption is undecidable, while that of unification is decidable.

(`const-variant-p` *clause1 clause2 const1 const2*)

should return `T` if *clause1*, constrained by *c*onst1, and *clause2*, constrained by *const2*, are variants of each other, and `'FAIL` otherwise.

(`const-instance-p` *clause1 clause2 const1 const2*)

should return `T` if *clause2* is an instance of *clause1*, that is, *clause1* is more general than *clause2* (constrained by *const1* and *const2* respectively). `'FAIL` should be returned otherwise.

(`const-subsumes-p` *clause1 clause2 const1 const2*)

should return `T` if *clause1* subsumes *clause2*, given their respective constraints *const1* and *const2*, and `FAIL` otherwise.

The corresponding functions `cls-variant-p`, `cls-instance-p` and `subsumes-p` for the unconstrained case are already available in FRAPPS (and H-FRAPPS). Note that all of these functions should, ideally, treat clauses as (unordered) sets of literals.

Once the above functions have been defined, forward and back subsumption work as described in the original FRAPPS manual, through the use of the functions `delete-fwd-sub-resolvents` and `deactivate-back-sub-nodes`.

The functions `get-ids-sub-by-node` and `get-ids-that-subsume-node` will also work as intended once the above functions have been defined.
However, the functions `get-ids-that-subsume-clause` and `get-ids-subsumed-by-clause` are now replaced by the following:

(`get-ids-subsumed-by-clause-const` *clause const*)

returns the ids corresponding to all the nodes in the derivation graph whose clauses (under their respective constraints) are subsumed by the *clause*, under the constraint *const*. Abbreviated to `giscc`.

(`get-ids-that-subsume-clause-const` *clause const*)

returns the ids corresponding to all the nodes whose clauses (under their respective constraints) subsume *clause*, under the constraint *const*. Abbreviated to `gitscc`.

(As in FRAPPS, all these subsumption procedures accept the keyword arguments [`:degree`], [`:weak`], [`:answer`] and [`:active`]. See the FRAPPS manual for more details.)

6.1 Literal Indexing

For efficiency, the literal indexing functions use the *unconstrained* (i.e. normal) unification procedure. Therefore, the following functions use standard unification in finding the clauses that satisfy the given requirement:

- `ret-clauses-cont-equal-lits`,

- `ret-clauses-cont-unifiable-lits`,
- `ret-clauses-cont-instance-lits` and
- `ret-clauses-cont-more-general-lits`

The user can then choose from the returned list of clauses those that really satisfy the given condition, under the new unifier. A generic implementation of these procedures using the user-defined unifier was deemed to be, in most cases, too inefficient to be of any practical use.

7 Session Management

The session management functions behave in very much the same way as in FRAPPS. However, the following points apply:

- H-FRAPPS session files are *not* compatible with FRAPPS, and vice-versa. (This is mainly because node-info structures are different in each.)
- The values of the global variables `*unifier*`, `*singleton*` and `*general-unifier*` are also saved in H-FRAPPS session files. However, the definitions of these and other functions that depend on the type of constraints used (mainly, the constraint-renaming and subsumption functions – see Sections 5.1 and 6) are *not* saved. The user will typically have a file containing the new definitions that is loaded into the Lisp environment *after* H-FRAPPS is initialized; these definitions are expected to be consistent with the global variables that indicate the type of unifier used.
- If constraints are structures, the user should redefine two additional function to guarantee that the session management procedures work under all implementations of Common Lisp:

  (`const-to-list` *const*) should take a constraint *const* as an argument and return a *list*, with no structures, containing the information in *const*.

  (`list-to-const` *list*) should reverse the above process, returning a constraint that is equivalent to the information "encoded" in *list*.

  The default for these two functions is the identity: H-FRAPPS assumes that constraints are lists. These functions can also be redefined if there is redundant information in constraint fields and users want to reduce the size of their session files.

8 Global Variables and User-Defined Functions

The following are the H-FRAPPS global variables not present in FRAPPS:

`*unifier*`

Identifies the function used to unify pairs of constrained literals. Default: `simple-unify` (see Section 4).

`*singleton*`

Should determine whether the unifier identified by `*unifier*` always returns a single unifier or not. Should be non-nil if more than one unifier can be produced, and `NIL` otherwise. Default: `NIL`.

`*general-unifier*`

Identifies the function used to unify sets of literals. Its default is `'general-unify`, which uses the literal-pair unifier given by `*unifier*`. Note: The value of this variable is only relevant when `*singleton*` is `NIL` (see Sections 4 and 5.2). Default: `NIL`

The following are some functions that users might want (or have) to redefine:

`(increment-constraint-subscripts` *const increment*`)`

Increments the subscripts of all the variables in the constraint *cost* by the amount specified by *increment* (see Section 5.1). By default, assumes that *const* is a list.

`(const-to-list` *const*`)` – `(list-to-const` *list*`)`

Used to translate constraints into lists and vice-versa, within the session management procedures (see Section 7). The default for both is the identity.

`(const-[variant|instance|subsumes]-p` *cls1 cls2 const1 const2*`)`

These three functions need to be defined if the user wants to perform the corresponding subsumption detection tests; they should return `T` if the clause *cls1*, constrained by *const2*, is a variant of, is an instance of, or subsumes the clause *cls2*, constrained by *const2*, and `FAIL` otherwise. (See Section 6.)

9 Example: Simple Sorted Unification

A simple constraint solving procedure is included along with H-FRAPPS: Sorted logic under the monomorphic tree restriction as presented in [Frisch, 1991].

We now briefly describe the functions that support these constraints using H-FRAPPS:[7] These can be loaded from H-FRAPPS with the function `(load-simple-sorts)`.

9.1 Defining Sorts

Sorts, like constants, are identified by Lisp atoms. (For example, `animal-1`, `odd` and `even` are valid sorts.) The global variable `*univ*` should always denote the universal (or "top") sort; its default is the atom `univ`.

Under the monomorphic tree restriction, the partial ordering on the given set of sorts can be represented as a tree, whose root is the universal sort: A sort S_1 is a descendant of a sort S_2 if and

[7]Thanks and acknowledgements go to Charlene Bloch, whose sorted unification procedures were the original basis for this code.

only if S_1 is a subsort of S_2. The subset relationships between the different sorts are defined using the following functions; these also, implicitly, define the set of sorts being used:

`(def-subsort` *sort1 sort2*`)`

Defines *sort1* to be a subsort of *sort2*. Assumes that *sort1* has not been defined as a subsort of anything else.

As a convenience to the user, the following function can be used to define the entire subtree in a single call:

`(def-subsort-list` *list*`)`

The argument list should be a list of pairs of the form `(`*x sort*`)`. For each one of these pairs, *sort* should be a single sort, while x can be a single sort or a list of sorts; all the sorts denoted by x are made subsorts of *sort*.

Finally, to add some flexibility to the order in which the different subsort relationships can be defined, the following function is provided:

`(move-sort` *sort1 sort2*`)`

Moves the subtree beginning at sort *sort1* so that it is an immediate subtree of that beginning at *sort2*. Assumes that both sorts have been defined and that *sort2* is not currently a subsort of *sort1*.

9.2 Defining the Sort of Terms

Under the tree restriction, a constant symbol, variable or term can only belong to a single most specific sort; that is, if a term t belongs to sorts S_1 and S_2, then either $S_1 \subseteq S_2$ or $S_2 \subseteq S_1$. The smallest sort to which a term t belongs is often said to be *the* sort of that term. Furthermore, monomorphism requires that all the terms constructed from a given function symbol belong to the same sort.

The sort of a variable is always its fourth symbol; that is, variables are lists of the form `(*var*` *name subscript sort*`)`. If a variable is a list of length 3 or its fourth element is `NIL`, the variable is said to be *unsorted*, and is implicitly taken to belong to the universal sort. Variables with the same name and subscript should *always* have the same sort.

The sort of non-variable terms is defined using the following function:

`(def-sortobject` *object sort*`)`

This function declares the constant or function symbol *object* to belong to the sort *sort*. If *object* is a function symbol f, this actually means that all the terms of the form $f(t_1, \ldots, t_n)$ are of sort *sort*.

This function should only be called once for each constant or function symbol. If the sort of a term is undefined, it is taken to be the universal sort.

The entire set of sort declarations can be reset using the function `(reset-sorts)`.

9.3 Defining Sorted Clauses

Note that since constraints are unary in this case, they are included directly in the variables. For the user's convenience the following function is provided:

`(s-def-clause` *clause* `[:sorts` *var-sort-list*`] [:user-field` *user-fld*`] [:support-set t])`

This function behaves exactly as `def-clause`, except that a constraint field is not specified and the unsorted variables in the clause *clause* are updated according to the sort constraints indicated by the keyword argument `[sorts:` *var-sort-list*`]`. The argument *var-sort-list* should be a list of pairs, each of the form `(`*var sort*`)`, indicating that the variable *var* should belong to the sort *sort*. In this way, the `?`*var-name* read-macro can still be used.

9.4 Unification and Subsumption for Sorted Clauses

The new unification procedure, called `sorted-unify`, finds the most general well-sorted (idempotent) unifier for two given terms, under the monomorphic tree sort definitions currently in effect, if one exists, and returns `FAIL` otherwise.[8] In this case, a singleton most general unifier is guaranteed to exist, so the `*singleton*` flag is set to `T`. The function's arguments and return values are basically in the same format as those of the `simple-unify` function described in Section 4.3, except that now variables contain sorts as their fourth element.

Note that the entire sort structure should be defined (using the functions described in the previous section) before any unification and/or deduction step is performed. Also, constraint fields and parameters are always `NIL`, since the constraints are built into the variables themselves.

Subsumption can be decided under this simple class of constraints in a relatively efficient way. The functions `const-variant-p`, `const-instance-p` and `const-subsumes-p` are redefined accordingly when the support for sorts is loaded into H-FRAPPS (See Section 6). Therefore, all the subsumption functions behave as expected. Note that those that require an input constraint should be called with this parameter set to `NIL`.

References

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

[Mitchell *et al.*, 1991] Michael K. Mitchell, Tomás E. Uribe, and Alan M. Frisch. *The Framework for Resolution-based Automated Proof Procedure Systems – FRAPPS 2.0 Users' Guide.* University of Illinois, Urbana, Illinois, September 1991.

[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.

[8] A well-sorted unifier is one that maps all variables into terms of the same or lesser sort.