Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL-90/30

ANL-90/30

DE91 001096

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

**A TUTORIAL ON THE CONSTRUCTION OF
HIGH-PERFORMANCE RESOLUTION/PARAMODULATION SYSTEMS**

by

Ralph Butler and Ross Overbeek*

Mathematics and Computer Science Division

September 1990

* Permanent Address: Computer Science Division, University of North Florida, 4567 St. John's Bluff Road South, Jacksonville, FL 32216

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

CONTENTS

Abstract	1
1. Introduction	1
2. Implementing the Basic Formula Database	1
2.1 Getting Started—Representing Clauses, Literals, and Terms	1
2.1.1 Representing Clauses	2
2.1.2 Representing Literals and Terms	4
2.1.3 Routines to Read and Write Clauses	11
2.2 Variables, Bindings, and Substitutions	16
2.3 Indexing Literals and Terms	28
3. A Simple Theorem Prover	33
4. Adding Support for Full Subsumption and Demodulation	45
4.1 Full Subsumption	48
4.2 Lexical Recursive Path Ordering	51
4.3 Demodulation	56
5. A High-Performance Implementation of the Indexing Routines	65
5.1 The Indexing Data Structure	66
6. Summary	68
References	69

A Tutorial on the Construction of High-Performance Resolution/Paramodulation Systems

by

Ralph Butler and Ross Overbeek

ABSTRACT

Over the past 25 years, researchers have written numerous deduction systems based on resolution and paramodulation. Of these systems, a very few have been capable of generating and maintaining a "formula database" containing more than just a few thousand clauses. These few systems were used to explore mechanisms for rapidly extracting limited subsets of "relevant" clauses. We have written this tutorial to reflect some of the best ideas that have emerged and to cast them in a form that makes them easily accessible to students wishing to write their own high-performance systems.

1. Introduction

Clause-based deduction systems have tended to fall into one of two categories, those that emphasized heuristics for making just the right set of deductions and those that emphasized the ability to generate and manage relatively large numbers of clauses. Clearly, the two goals are not incompatible, but the effort required to construct a system capable of successfully performing operations on large sets of clauses was typically viewed by many researchers as a distraction. Admittedly, the effort to produce a high-performance system might have been substantial ten years ago. With this tutorial, we demonstrate that you can now produce an effective system with minimal effort and resources.

In the following pages, we construct a fairly powerful theorem prover. We proceed in small steps, including aspects of the "folk wisdom" that have been omitted in more formal reports. Our intent is not simply to familiarize you with the particular system that we construct. Rather, we present an approach for constructing such systems so that you may easily construct a system tailored to your particular needs.

The system that we present is a standard, clause-based resolution/paramodulation system. While we consider it to be a fairly basic system, it is powerful enough to be capable of deriving proofs that the vast majority of systems described in the literature cannot. Hence, we do view it as a "serious" system. We offer it as a good starting point for anyone who wishes to extend automated deduction within the classic clause-based paradigm.

2. Implementing the Basic Formula Database

2.1. Getting Started—Representing Clauses, Literals, and Terms

One of the first decisions that you must make when implementing an automated deduction system is how you wish to represent the formulas that you intend to manipulate. This choice should be made fairly carefully, and then the basic objects should be implemented by using data abstraction techniques. To the extent that you do effectively implement a set of abstract data types, you should be able to alter your base representation without impacting higher layers of code. On the other hand, you will find us (very occasionally) violating principles of data abstraction to get more performance. We will point out such

transgressions as they occur.

2.1.1. Representing Clauses

We have implemented clauses as a structure that includes two sets of literals and an *id*. This representation would not be desirable for systems that operate on clauses in which literals are thought of as ordered (since you would want a single set, in that case). In our opinion, the choice of the format of a clause is not one of the central decisions; it is usually far less important than deciding on the representation for literals and terms. Anyway, our choice works nicely for our system:

```
#define MAX_LTS_CL      6
typedef struct clause   *CLAUSE;
typedef struct lt       *LT;

struct clause {
    struct clause *link;      /* used to put clause in lists */
    int id;                  /* unique id; -1 unless set */
    char n,p;                /* n is # negative literals;
                             p is # positive literals */
    LT lits[MAX_LTS_CL];     /* pointers to literals */
};
```

We think of a clause as having an *id*, which is -1 until we decide to keep the clause; then it will be assigned a unique value. To support unique *ids*, we include the global variable

```
int xx_next_id = 1;
```

which is always set to the next available clause *id*. The three fields *n*, *p*, and *lits* are used to record the literals in the clause. The array *lits* contains pointers to the negative literals, followed by pointers to the positive literals.

The remaining field, *link*, requires some discussion. In our system, we explicitly allocate and deallocate each type of dynamically allocated structure. Most programmers familiar with a higher-order language (such as Prolog or Lisp) will wonder why we do not simply use a garbage collector. The relative merits of garbage collection versus explicit allocation/deallocation have been vigorously debated in many contexts. It is true that, if you are not careful, as much as 75% of execution time can be expended in memory management; if you are careful, however, that percentage should be negligible (under 2%). We include a global variable

```
CLAUSE *avl_cl = NULL;
```

and we use the following routine to allocate such nodes:

```
CLAUSE alloc_cl()
{
    CLAUSE c;
    int i;

    if (!avl_cl)
    {
        /* allocate memory for 100 clause nodes */
        if ((avl_cl = (CLAUSE) malloc(sizeof(struct clause)*100))
            == NULL)
        {
```

```

        printf("malloc error in alloc_cl: %d bytes\n",
               sizeof(struct clause)*100);
        exit(1);
    }

    /* put 100 unused nodes in available-node list */
    for (i=0; i < 99; i++)
        (avl_cl + i)->link = avl_cl+(i+1);
    (avl_cl + 99)->link = NULL;
}
c = avl_cl;
avl_cl = c->link;
return(c);
}

```

Since we will have a similar routine for each type of node we introduce, you should make sure that it makes sense to you.

As we mentioned, it's a good idea to hide the actual representation from the higher layers of code. In the case of clauses, the hiding is implemented with the following macros and subroutines:

```

#define num_neg_lits(C) ((C)->n)
#define num_pos_lits(C) ((C)->p)
#define nth_neg(C,N) ((C)->lits[(N)])
#define nth_pos(C,N) ((C)->lits[(N)+((C)->n)])

#define assign_id(C) ((C)->id = xx_next_id++)
#define cl_id(C) ((C)->id)

/* The basic clause constructor; build a clause, assuming
   that the literals that make up the clause have been
   constructed. Here the negative literals are referenced
   by the array of pointers neg_lits. The array is "null
   terminated". That is, if there are i negative literals,
   then neg_lits will contain i pointers to those literals,
   followed by a NULL pointer. Similarly for pos_lits.
*/
CLAUSE make_clause(neg_lits,pos_lits)
LT neg_lits[],pos_lits[];
{
    CLAUSE c;
    int i,j;

    c = alloc_cl();
    c->id = -1;

    for (i=0; neg_lits[i] && (i < MAX_LTS_CL); i++)
        c->lits[i] = neg_lits[i];
    c->n = i;

    for (j=0; pos_lits[j] && ((i+j) < MAX_LTS_CL); j++)
        c->lits[i+j] = pos_lits[j];
    c->p = j;

    if (((i + j) == MAX_LTS_CL) && (neg_lits[i] || pos_lits[j]))
    {
        fprintf(stderr,"Too many literals in clause;");
        fprintf(stderr," increase MAX_LTS_CL and recompile\n");
        exit(1);
    }
}

```

```
    }
    return(c);
}

dealloc_cl(c)
CLAUSE c;
{
    int i;

    for (i=0; i < num_neg_lits(c); i++)
        dealloc_lt(nth_neg(c,i));    /* defined later */

    for (i=0; i < num_pos_lits(c); i++)
        dealloc_lt(nth_pos(c,i));

    c->link = avl_cl;
    avl_cl = c;
}
```

Later, we will include routines to copy a clause and to rename variables in a clause. For now, the above definitions and routines are enough to get started. You should consult the Appendix for a description of the interface to the abstract data type *CLAUSE* (along with the interfaces to the other abstract data types that we will implement).

2.1.2. Representing Literals and Terms

The choice of how to represent literals and terms is fairly critical. There are numerous tradeoffs between simplicity, flexibility, and performance. Issues relating to structure-sharing, compilation, and indexing all come into play. The choice that we have made offers, in our opinion, good performance and relative simplicity.

Since we have chosen to view literals as unsigned (i.e., the sign is conveyed by which set the literal occurs in within the clause), literals and terms have exactly the same structure. Literals/terms are typed as *Var*, *Constant*, or *Complex*. Variables are thought of as *V0*, *V1*, *V2*,.... Complex literals/terms are naturally thought of as trees. However, as Jim Christian [1] demonstrated, there is also an advantage in viewing the literal/term as a "linear string of symbols." To enable a term to be viewed from either perspective, you can use a structure such as

```
#define Var          0
#define Constant    1
#define Complex      2

struct lt {
    LT prev,next,end;
    char type; /* Var, Constant, or Complex */
    short val; /* Var -> variable number;
                Constant -> symbol id + arity of 0
                Complex -> symbol id of functor + arity

                For Constant and Complex lts,
                the val field includes both
                the symbol id and the arity (0 for
                Constants). The arity is in the right
                4 bits; the symbol id is computed
                by shifting val right 4 bits. */
    int var_occ; /* bitmap of variables that occur in lt.
                  If Vi occurs within the complete subterm
                  beginning at this lt, bit i will be
```

```
set to 1 (thinking of the rightmost bit  
as 0). */
```

```
};
```

There is one *lt* node per symbol in a literal/*lt*. For example, to represent

```
f(a, f(v0, b))
```

we would use five *lt* nodes. The *prev* and *next* fields are used to move through the literal/term in the linear representation. That is, if *p1* were a pointer to the *lt* node representing *a*, then *p1*->*next* would reference the node representing the *f* in the embedded complex term. For any *lt* node, we will speak of the "complete subterm" beginning at that node. For a *Var* or *Constant*, the complete subterm is just the single node; for a *Complex* node, the complete subterm is just the entire literal/term that begins with the *Complex* node. The *end* field points to the last symbol of the complete subterm that begins at the node. Thus, since *p1* points to a node of type *Constant*, *p1*->*end* is just *p1*. On the other hand, *p1*->*next*->*end* would point to the node representing *b*. This situation is all depicted in the following diagram:

Representation of $f(a, f(V0, b))$

The *type* and *val* fields of an *lt* are used to capture one symbol. We convert symbols to "symbol ids," which are just subscripts in a symbol table. Thus, the actual symbols to represent function and predicate symbols are referenced only by input/output routines that have access to the symbol table. The remainder of the code references function/predicate symbols as fairly small integers. Since we find it convenient to keep the arity around, we actually set the *val* field to include both the symbol id and the arity of the symbol.

The *var_occ* field records the variables that occur within the complete subterm beginning at the node. For example, if the *var_occ* field is 0, then the complete subterm is ground. Later, we will see that this field can be used for a number of purposes, including a relatively fast "occurs check." A disadvantage of using such a field centers on the need to keep variable numbers in the range 0 to *MAXVAR* and to maintain the bit vector. We require that the bits be set properly in any literal/term before attempting to unify it with another literal/term. A literal/term with the variable numbers in the acceptable range (and with the bits set properly in *var_occ*s) are called *normalized*; other literal/terms are called *unnormalized*. Given any clause containing unnormalized literals, the clause can be normalized by using the routine *rename_cl*, which is given below.

Just as with *CLAUSE* nodes, we include a global variable

```
LT *avl_lt = NULL;
```

and we use the following routine to allocate *lt* nodes:

```
LT alloc_lt()
{
    LT p;
    int i;

    if (!avl_lt)
    {
        if ((avl_lt = (LT) malloc(sizeof(struct lt) * 1000)) == NULL)
        {
            printf("malloc error in alloc_lt: %d bytes\n",
                sizeof(struct lt)*1000);
            exit(1);
        }
        for (i=0; i < 999; i++)
            (avl_lt + i)->next = avl_lt+(i+1);
        (avl_lt + 999)->next = NULL;
    }
    p = avl_lt;
    avl_lt = p->next;
    return (p);
}
```

The higher-level code uses the following three constructors to build *lt* nodes (*make_constant* to make constants, *make_var* to make nodes representing variables, and *make_complex* to make structures). It uses *dealloc_lt()* to explicitly deallocate nodes (note that the complete subterm is deallocated).

```
/* the caller is responsible for converting a symbol to
   the corresponding integer, which we call a "symbol id"
*/
LT make_constant(sym_id)
int sym_id;
{
    LT p;

    p = alloc_lt();
    p->end = p;
    p->type = Constant;
    p->val = sym_id << 4;
    return (p);
}
```

```

LT make_var(vn)
int vn;
{
  LT p;

  p = alloc_lt();
  p->end = p;
  p->type = Var;
  p->val = vn;
  return(p);
}

/* this constructor takes the symbol id of the functor,
   along with a NULL terminated array of pointers to the
   arguments.
*/
LT make_complex(func,args)
int func;
LT args[];
{
  LT p1,p2,p3;
  int i;

  p1 = alloc_lt();
  p1->type = Complex;
  p1->prev = NULL;

  for (i=0,p2=p1; (p3 = args[i]); i++)
  {
    p3->prev = p2;
    p2->next = p3;
    p2 = p3->end;
  }
  p1->val = (func << 4) | i;
  p1->end = p2;
  p2->next = NULL;
  return(p1);
}

/* this deallocation routine frees the complete subterm */
dealloc_lt(p)
LT p;
{
  p->end->next = avl_lt;
  avl_lt = p;
}

```

A simple version of the *rename_cl* routine (which normalizes all of the literals in the clause) is as follows:

```

int rename_cl(c)
CLAUSE c;
{
  int vtab[200];
  int i;

  vtab[0] = -1;
  for (i=0; i < num_neg_lits(c); i++)
  {

```

```
        rename_bod(nth_neg(c,i),vtab);
    }
    for (i=0; i < num_pos_lits(c); i++)
    {
        rename_bod(nth_pos(c,i),vtab);
    }

    for (i=0; vtab[i] != -1; i++)
    ;
    return(i);
}

int rename_bod(term,vtab)
LT term;
int vtab[];
{
    int i;
    LT al;
    TPOS pl;

    if (type_lt(term) == Var)
    {
        for (i=0;
            (i < 200+1) && (vtab[i] != -1) && (vtab[i] != vnum(term));
            i++)
        ;
        if (i == 200+1)
        {
            printf("Tried to rename a term with too many variables exiting0);
            exit(1);
        }

        if (vtab[i] == -1)
        {
            vtab[i] = vnum(term);
            vtab[i+1] = -1;
        }
        term->val = i;
        term->var_occ = 1 << term->val;
        return(term->var_occ);
    }
    else if (type_lt(term) == Complex)
    {
        term->var_occ = 0;
        for (al=first_arg(term,&pl); al; al=next_arg(&pl))
            term->var_occ |= rename_bod(al,vtab);
        return(term->var_occ);
    }
    else
    {
        term->var_occ = 0;
        return(0);
    }
}
```

Note that *rename_cl* returns the number of distinct variables in the clause. It is up to the caller to verify that this value is less than or equal to *MAXVAR*. The routine can handle clauses with up to 200 distinct variables; if you try to normalize a clause with more variables, the routine will force an abnormal

termination of the whole program.

Just as with the data type *CLAUSE*, we wish to use a set of macros and subroutines to define the operations that can be performed on *LT*'s. The basic access operations are defined by the following macros:

```
#define type_lt(L)      ((L)->type)
#define symbol_id(L)    ((L)->val >> 4)
#define vnum(L)         ((L)->val)
#define functor(L)      (((L)->val) >> 4)
#define arity(L)        (((L)->val) & 15)
#define vars_in(L)      ((L)->var_occ)
```

We also need operations that allow us to traverse either the symbols in a literal/term or the immediate arguments of a literal/term (both types of traversal are useful). To support such types of traversal, we write "first/next" routines:

```
#define BOOL    int
#define TRUE    1
#define FALSE   0

/* a TPOS supports "first/next" argument, while
   an SPOS supports "first/next" symbol
*/
typedef struct tpos      TPOS;
typedef struct spos      SPOS;

struct tpos {
    LT end;
    LT last;
};

struct spos {
    LT end;
    LT last;
};

/* first_arg/next_arg are used to go through arguments
   of a complex term
*/
LT first_arg(trm,p)
LT trm;
TPOS *p;
{
    p->end = trm->end->next;
    return(p->last = trm->next);
}

LT next_arg(p)
TPOS *p;
{
    LT r;

    if ((r = p->last->end->next) != p->end)
    {
        p->last = r;
        return(r);
    }
}
```

```
    }
    else
        return (NULL);
}

/* first_sym/next_sym go through the symbols in a
   complex term. first_sym returns the symbol id of the
   functor. Subsequent calls to next_sym return the
   type and value for the symbols that occur in the term
*/

LT first_sym(trm,p)
LT trm;
SPOS *p;
{
    p->end = trm->end;
    return(p->last = trm);
}

LT next_sym(pl)
SPOS *pl;
{
    LT p;

    if (pl->last != pl->end)
    {
        p = pl->last->next;
        return(pl->last = p);
    }
    else
        return (NULL);
}
```

These routines are based on the notion that the "first" operation establishes a "current position." Subsequent "next" operations move ahead from the current position. We frequently find ourselves creating a data structure to record a "current position" (i.e., a structure to record some notion of "local state"), along with first/next operations. Hence, we suggest that you become familiar with this style.

2.1.3. Routines to Read and Write Clauses

To get things into perspective, let us write a complete program that reads and prints clauses. It will force us to get things organized.

We think of the routines that support the basic data types (so far, we have talked about *CLAUSES* and *LTs*) as "the formula database." Our exact reasons will become clear when we talk about the remaining data types. We propose that you put all of the structure definitions, routine declarations, and macro definitions for the basic data types into a file called *fdb.h* and the code for the support routines into *fdb.c*. Once that is done, our main program becomes something like

```
#include "fdb.h"
#include <stdio.h>

/* This theorem prover takes a set of clauses as input.
   Each clause is on a single line. Rather than give a
   grammar, let me just give a few examples from the P formulation
   of the axioms for a group:
```

```
() ((P V0 e V0))
((P V0 V1 V2) (P V1 V3 V4) (P V0 V4 V5)) ((P V2 V3 V5))
() ((P V0 V1 (f V0 V1)))
((P b a c)) ()
```

That is, the clause is represented as two lists of literals; the first being the negative literals and the second the positive. Variable are of the form Vi. Constants and functors are single characters (other than V).

This is a pretty awful input format, but it will do for purposes of illustration (and it is easy to parse!). Our routines do not do proper error checking -- they really are only for illustration.

*/

```
char symtab[100];
```

```
main()
```

```
{
  CLAUSE read_list();
  CLAUSE clauses;
  CLAUSE cl;
```

```
  symtab[0] = 0; /* start with an empty symbol table */
```

```
  clauses = read_list();
```

```
  printf("clauses:\n");
  print_list(closures);
```

```
}
```

```
CLAUSE read_list()
```

```
{
  CLAUSE read_clause();
  CLAUSE clist = NULL;
  CLAUSE next_cl,tmp;
```

```
  while (next_cl = read_clause())
```

```
  {
    next_cl->link = clist;
    clist = next_cl;
  }
```

```
  return(clist);
```

```
}
```

```
CLAUSE read_clause()
```

```
{
  LT neg[MAX_LTS_CL+1],pos[MAX_LTS_CL+1]; /* MAX_LTS_CL is in fdb.h */
  CLAUSE cl;
```

```
  if (getchar() == '(')
```

```
  {
    read_literals(neg);
```

```
    while (getchar() != '(')
```

```
    ;
    read_literals(pos);
```

```
        cl = make_clause(neg,pos);
        assign_id(cl);

        while (getchar() != '\n')
            ;
    }
    else
        cl = NULL;

    return(cl);
}

read_literals(lits)
LT lits[];
{
    int i;
    LT read_lt();
    char c;

    while ((c = getchar()) == ' ')
        ;

    for (i=0; (c == '('); i++)
    {
        lits[i] = read_lt();

        while ((c = getchar()) == ' ')
            ;
    }
    lits[i] = NULL;
}

LT read_lt()
{
    int func,vn;
    int nargs = 0;
    LT args[20];
    char c;

    c = getchar();
    func = get_sym_entry(c);
    getchar();

    c = getchar();
    while (c != ')')
    {
        if (c == '(')
            args[nargs++] = read_lt();
        else if (c == 'V')
        {
            vn = getchar() - '0';    /* max 10 variables/clause */
            args[nargs++] = make_var(vn);
        }
        else
            args[nargs++] = make_constant(get_sym_entry(c));

        if ((c = getchar()) == ' ')
            c = getchar();
    }
}
```

```
    args[nargs] = NULL;
    return(make_complex(func,args));
}

print_list(clist)
CLAUSE clist;
{
    while (clist)
    {
        print_cl(clist);
        clist = clist->link;
    }
    printf("\n");
}

print_cl(cl)
CLAUSE cl;
{
    int i;
    LT lt;

    printf("id=%d neg=(",cl_id(cl));
    for (i=0; i < num_neg_lits(cl); i++)
    {
        print_lt(nth_neg(cl,i));
        if (i < num_neg_lits(cl)-1)
            printf(",");
    }
    printf(") pos=(";
    for (i=0; i < num_pos_lits(cl); i++)
    {
        print_lt(nth_pos(cl,i));
        if (i < num_pos_lits(cl)-1)
            printf(",");
    }
    printf(")\n");
}

print_lt_nl(t)
LT t;
{
    print_lt(t);
    printf("\n");
}

print_lt(trm)
LT trm;
{
    TPOS p;
    LT t;

    if (type_lt(trm) == Var)
        printf("V%d",vnum(trm));
    else if (type_lt(trm) == Constant)
        printf("%c",syntab[symbol_id(trm)]);
    else
    {

```



```
printf("%c(", symtab[functor(trm)]);
t = first_arg(trm, &p);
print_lt(t);
while (t = next_arg(&p))
{
    printf(",");
    print_lt(t);
}
printf(")");
}

int get_sym_entry(c)
char c;
{
    int i;

    for (i=0; symtab[i] && (symtab[i] != c); i++)
        ;

    if (symtab[i] != c)
    {
        symtab[i] = c;
        symtab[i+1] = 0;
    }
    return(i);
}
```

The makefile that you should use to glue things together should be as follows:

```
#
# We assume that you put your test program in testio.c.
# Further, we assume that you are using the GNU C compiler
# (you should; it's really great); if you are not, just change
# 'gcc' to 'cc' below.

testio: testio.o fdb.o
        gcc -g -o testio testio.o fdb.o

CFLAGS = -g

.c.o:      *.c
        gcc -c $(CFLAGS) *.c
```

Exercises:

1. Successfully make this program read and print clauses. Then convert the following clauses to the format required for the input routines:

Clause 1: $\neg p(x) \mid \neg p(i(x,y)) \mid p(y)$.

Clause 2: $p(i(i(i(x,y),z),i(i(z,x),i(u,x))))$.

Clause 3: $\neg p(i(i(i(a,b),a),a))$.

Make sure that your program can successfully read and print these clauses.

2. Some theorem provers use a more readable format for clauses. For example, you might wish to read in clauses as written above, or to read clauses in something like a Prolog format. Write a grammar for the format that you think will be appropriate for your system; write input and output routines, and debug the routines using a modified version of the program given above.

2.2. Variables, Bindings, and Substitutions

One central issue related to implementing deduction systems involves representing substitutions. Many textbooks talk about creating "variants" of clauses, with variables renamed to avoid duplicates between distinct clauses. In practice, variants are never created, since the cost of copying the clause is quite unnecessary.

There are many ways to handle the instantiation of variables. We choose to introduce the notion of "substitution," which acts as a binding environment for variables. When one is sure that the variables in a literal/term are not bound, a sequence of *LT* nodes may be used to represent the literal/term. However, if variables may have been bound (or perhaps will be bound), we represent a literal/term as a 2-tuple—the sequence of *LT* nodes and a *SUBSTITUTION* to record bindings. A binding assigns an image to a variable; since the image is a term that may contain bound variables, an image itself must be a 2-tuple.

We think of literals/terms as either *normalized* or *unnormalized*. A normalized term may contain variables only in the range 0-MAXVAR, where MAXVAR is set at compile time. Only normalized literals/terms can have associated binding environments (i.e., substitutions). Here is our definition of a substitution:

```
#define MAXVAR          10

typedef struct substitution *SUBSTITUTION;

struct sub_entry {
    LT image;
    struct substitution *env;
};

struct substitution {
    struct substitution *link;
    struct sub_entry sub_ent[MAXVAR];
    int offset;           /* used by apply */
    int bound;           /* bitmap of bound variables */
};
```

The field *link* is again used for allocation/deallocation. With *CLAUSEs* and *LTs*, we used constructors to build occurrences of the appropriate structure (so, higher-level routines would never actually call the allocation routine—only the constructors would). However, in the case of *SUBSTITUTIONs*, the higher level routines explicitly allocate structures with

```
SUBSTITUTION alloc_sub()
{
    SUBSTITUTION sub;
    int i;

    if (!avl_sub)
    {
        if ((avl_sub=(SUBSTITUTION)malloc(sizeof(struct substitution)*20)) == NULL)
        {
            printf("malloc error in alloc_sub: %d bytes\n",
                sizeof(struct substitution)*20);
        }
    }
}
```

```
        exit(1);
    }
    for (i=0; i < 19; i++)
    {
        (avl_sub + i)->link = avl_sub+(i+1);
    }
    (avl_sub + 19)->link = NULL;
}
sub = avl_sub;
avl_sub = sub->link;
init_sub(sub);

return(sub);
}

init_sub(sub)
SUBSTITUTION sub;
{
    int i;

    sub->bound = 0;
    for (i=0; i < MAXVAR; i++)
        sub->sub_ent[i].image = NULL;
}
```

Deallocation of substitutions can be accomplished by using

```
dealloc_sub(sub)
SUBSTITUTION sub;
{
    sub->link = avl_sub;
    avl_sub = sub;
}
```

A variable can be bound by adding an entry to a substitution. In many cases, it is useful to keep a record of the bindings introduced by an operation like unification. Such a record is called a *trail*, and it is used to undo exactly those bindings introduced by a single unification (a substitution may be built up over several unifications; for example, consider how a hyperresolvent is determined by a sequence of unifications). A trail is represented by the following types:

```
typedef struct trail    TRAIL;

struct trail_ent {
    int v;                /* subscript of bound variable */
    SUBSTITUTION s;       /* binding env used to record image */
};

struct trail {
    struct trail_ent tent[2*MAXVAR];
    int nxt;              /* subscript of first unused entry */
};
```

Note that the type *TRAIL* is not a pointer, but rather the actual structure. The operation of binding a variable and recording the binding in a trail is accomplished by the following routine:

```
bind(vn, sub, tr, image_trm, image_sub)
int vn; /* subscript of bound integer */
SUBSTITUTION sub; /* make entry in this binding env */
TRAIL *tr; /* pointer to trail that must be updated */
LT image_trm; /* bind variable to this LT */
SUBSTITUTION image_sub; /* env for variables in image_trm */
{
    sub->sub_ent[vn].image = image_trm;
    sub->sub_ent[vn].env = image_sub;
    sub->bound |= 1 << vn;

    tr->tent[tr->nxt].v = vn;
    tr->tent[tr->nxt++].s = sub;
}
```

Thus,

```
bind(i, sub, tr, trm, sub2)
```

would bind *V_i* in *sub* to (*trm*, *sub2*), recording the binding in the trail *tr*. We note that it is a common technique to rewrite such small routines as macros to attain higher performance. This is justified only when the routine actually consumes a fair percentage of the execution time. Furthermore, the better C compilers (e.g., GNU C) have the option of generating in-line code for small routines. In any event, here is an optimized version of *bind* implemented as a macro:

```
#define bind(VarNum, Sub, Trail, Image, ImgEnv)
{
    struct sub_entry *se;
    struct trail_ent *te;

    se = (Sub)->sub_ent + (VarNum);
    te = (Trail)->tent + ((Trail)->nxt++);
    se->image = Image;
    se->env = ImgEnv;
    te->s = Sub;
    te->v = VarNum;
    (Sub)->bound |= 1 << VarNum;
}
```

The following little routine can be used to "undo" the bindings recorded in a trail (reinitializing the trail to empty).

```
undo_bindings(tr)
TRAIL *tr;
{
    SUBSTITUTION s1;
    int i;

    while (tr->nxt > 0)
    {
        tr->nxt--;
        s1 = tr->tent[tr->nxt].s;
        i = tr->tent[tr->nxt].v;
        s1->sub_ent[i].image = NULL;
        s1->bound &= ~(1 << i);
    }
}
```

)
)

In the remainder of this section, we consider how to implement three critical operations:

1. A *match* operation, commonly used in implementing subsumption and demodulation, to see whether one term is an instance of another.
2. A *unify* operation, used to compute a most general common instance of two literals or terms.
3. An *apply* operation, used to apply the bindings recorded in one or more substitutions to a literal or term, producing a new copy in which all variables that have been bound have been replaced by their bindings.

Here is the simplest of the three operations, the *match*.

```

BOOL match(lt1, sub1, lt2, tr)
LT lt1, lt2;
SUBSTITUTION sub1;
TRAIL *tr;
{
  int i;

  /* initialize the trail */
  t1->nxt = 0;

  if (match_bod(lt1, sub1, lt2, tr))
    return(TRUE);
  else
  {
    /* the match attempt may have bound some variables; we must
       undo those before exiting */
    undo_bindings(tr);
    return(FALSE);
  }
}

BOOL match_bod(lt1, sub1, lt2, tr)
LT lt1, lt2;
SUBSTITUTION sub1;
TRAIL *tr;
{
  LT end, lt1_d;
  BOOL eq_trm();

  end = lt1->end->next;

  while (lt1 != end)
  {
    if (type_lt(lt1) == Var)
    {
      if (lt1_d = sub1->sub_ent[vnum(lt1)].image)
      {
        /* the variable has been bound to the term pointed
           to by lt1_d. Hence, for the match to succeed,
           lt1_d and lt2 must point to totally identical
           terms.
          */
        if (!eq_trm(lt1_d, lt2))
          return FALSE;
      }
    }
  }
}
```

```
        else
        {
            lt1 = lt1->next;
            lt2 = lt2->end->next;
        }
    }
    else
    {
        /* the variable is unbound, so bind it to the
           corresponding term (lt2)
        */
        bind(vnum(lt1), sub1, tr, lt2, NULL);
        lt1 = lt1->next;
        lt2 = lt2->end->next;
    }
}
else
{
    if ((lt1->type != lt2->type) || (lt1->val != lt2->val))
        return(FALSE);
    else
    {
        lt1 = lt1->next;
        lt2 = lt2->next;
    }
}
return(TRUE);
}

/* returns TRUE iff the two terms are totally identical */
BOOL eq_trm(lt1, lt2)
LT lt1, lt2;
{
    LT end;

    for (end=lt1->end->next; lt1 != end; lt1=lt1->next, lt2=lt2->next)
        if ((type_lt(lt1) != type_lt(lt2)) || (lt1->val != lt2->val))
            return(FALSE);

    return(TRUE);
}
```

The problem of actually computing a unifier of two terms is a bit more complex. Here is a simple version of the required code:

```
BOOL unify(lt1, sub1, lt2, sub2, trail)
LT lt1, lt2;
SUBSTITUTION sub1, sub2;
TRAIL *trail;
{
    register int i;

    trail->nxt = 0;
    if (unify_bod(lt1, sub1, lt2, sub2, trail))
        return(TRUE);
}
```

```
else
{
    undo_bindings(trail);
    return(FALSE);
}

/* the following macro "dereferences an (Lt,Sub)". That is,
   if Lt points to a variable, a check will be made to
   see if it is bound in Sub; if it is, Lt and Sub will be
   updated to the image, and the image will be dereferenced.
*/
#define Deref(Lt,Sub)
{
    LT xlt1;

    while ((type_lt(Lt) == Var) &&
           && Sub != NULL
           && (xlt1 = Sub->sub_ent[vnum(Lt)].image))
    {
        Sub = Sub->sub_ent[vnum(Lt)].env;
        Lt = xlt1;
    }
}

BOOL unify_bod(lt1, sub1, lt2, sub2, trail)
LT lt1, lt2;
SUBSTITUTION sub1, sub2;
TRAIL *trail;
{
    LT a1, a2;
    TPOS p1, p2;

    Deref(lt1, sub1);
    Deref(lt2, sub2);

    if (type_lt(lt1) == Var)
    {
        if ((type_lt(lt2) == Var) && (lt1->val == lt2->val) && sub1 == sub2)
            return(TRUE);
        else if (occurs(vnum(lt1), sub1, lt2, sub2))
            return(FALSE);
        else
        {
            bind(vnum(lt1), sub1, trail, lt2, sub2);
            return(TRUE);
        }
    }
    else if (type_lt(lt2) == Var)
    {
        if (!occurs(vnum(lt2), sub2, lt1, sub1))
        {
            bind(vnum(lt2), sub2, trail, lt1, sub1);
            return(TRUE);
        }
        else
            return(FALSE);
    }
    else

```

```
(
  if ((type_lt(lt1) != type_lt(lt2)) ||
      (lt1->val != lt2->val))
    return(FALSE);
  else
  (
    if (type_lt(lt1) == Constant)
      return(TRUE);
    else
    (
      for (a1=first_arg(lt1,&p1),a2=first_arg(lt2,&p2);
          a1;
          a1=next_arg(&p1), a2=next_arg(&p2))
      (
        if (!unify_bod(a1,sub1,a2,sub2,trail))
          return(FALSE);
      )
      return(TRUE);
    )
  )
)
```

/* This routine returns TRUE iff the variable vn (sub1) occurs
(t,sub2). The test is done by utilizing the var_occ field
that is kept in LTs, along with the 'bound' field kept in
substitutions. We think that it is probably far faster
than searching the entire LT for an occurrence.

*/

BOOL occurs(vn,sub1,t,sub2)

int vn;

LT t;

SUBSTITUTION sub1,sub2;

```
(
  struct occ_stack {
    int var_occ;
    SUBSTITUTION sub;
  } os[50], *top;
```

SUBSTITUTION s,s1;

int vo;

LT img;

int i,j;

int vnbit;

```
if (!t->var_occ || ((sub1 != sub2) && (!sub2->bound)))
  return(FALSE);
```

vnbit = 1 << vn;

```
if ((sub1 == sub2) && (vnbit & t->var_occ))
  return(TRUE);
```

os[0].var_occ = t->var_occ;

os[0].sub = sub2;

top = os+1;

while (top-- != os)

```
(
  vo = top->var_occ;
  s = top->sub;
```



```

for (i=0; vo; i++, vo >>= 1)
{
    if (vo & 1)
    {
        if ((img = s->sub_ent[i].image) &&
            (j = img->var_occ))
        {
            if (((s1 = s->sub_ent[i].env) == sub1) &&
                (j & vnbit))
                return(TRUE);
            else
            {
                top->var_occ = j;
                top->sub = s1;
                top++;
            }
        }
    }
}
return(FALSE);
}

```

The performance of the unification code can be improved significantly by two techniques:

1. When processing two COMPLEX terms, the arguments are processed by making a recursive invocation of *unify_bod*. The recursive call should be avoided in all cases in which at least one of the arguments is not COMPLEX.
2. The *unify_bod* routine, which is recursive, assumes that the two input terms could be of any type. However, if the recursive call is made only when both terms are COMPLEX and have identical functors, it is more efficient to introduce a new recursive routine, *unify_comp*, in which the input literals/terms are known to be COMPLEX with matching functors.

Such detailed optimizations are justified only for routines that represent significant percentages of the overall execution time. Since unification is such an area, the optimizations are probably worth introducing, giving the following code:

```

BOOL unify_bod(lt1, sub1, lt2, sub2, t1)
LT lt1, lt2;
SUBSTITUTION sub1, sub2;
TRAIL *t1;
{
    Deref(lt1, sub1);
    Deref(lt2, sub2);
    if (type_lt(lt1) == Complex)
    {
        if (type_lt(lt2) == Complex)
        {
            if (lt1->val == lt2->val)
                return(unify_comp(lt1, sub1, lt2, sub2, t1));
            else
                return(FALSE);
        }
        else if (type_lt(lt2) == Var)
        {
            if (occurs(vnum(lt2), sub2, lt1, sub1))
                return(FALSE);
            else

```

```
        {
            bind(vnum(lt2), sub2, t1, lt1, sub1);
            return(TRUE);
        }
    }
    else
        return(FALSE);
}
else if (type_lt(lt1) == Var)
{
    if ((type_lt(lt2) == Var) && (lt1->val == lt2->val) && sub1 == sub2)
        return(TRUE);
    else if (occurs(vnum(lt1), sub1, lt2, sub2))
        return(FALSE);
    else
    {
        bind(vnum(lt1), sub1, t1, lt2, sub2);
        return(TRUE);
    }
}
else
{
    if (type_lt(lt2) == Constant)
    {
        return(lt1->val == lt2->val);
    }
    else if (type_lt(lt2) == Var)
    {
        bind(vnum(lt2), sub2, t1, lt1, sub1);
        return(TRUE);
    }
    else
        return(FALSE);
}
}
```

/* Here, we go even further. We avoid recursive calls to unify_comp by introducing our own "problem stack". These optimizations are a pain, but they do make a difference.

```
*/
BOOL unify_comp(lt1, sub1, lt2, sub2, t1)
LT lt1, lt2;
SUBSTITUTION sub1, sub2;
TRAIL *t1;
{
    struct mstack {
        LT a1, a2, end;
        SUBSTITUTION s1, s2;
    } ms[50], *top;
```

```
    LT a1, a2, end, a1_n, a2_n;
    SUBSTITUTION s1, s2, s1_d, s2_d;
```

```
    top = ms+1;
    ms[0].a1 = lt1->next;
    ms[0].a2 = lt2->next;
    ms[0].end = lt1->end->next;
    ms[0].s1 = sub1;
    ms[0].s2 = sub2;
```

```
while (top-- != ms)
{
    a1 = top->a1;
    a2 = top->a2;
    end = top->end;
    s1 = top->s1;
    s2 = top->s2;

    while (a1 != end)
    {
        s1_d = s1;
        s2_d = s2;
        a1_n = a1->end->next;
        a2_n = a2->end->next;
        Deref(a1,s1_d);
        Deref(a2,s2_d);
        if (type_lt(a1) == Complex)
        {
            if (type_lt(a2) == Complex)
            {
                if (a1->val == a2->val)
                {
                    top->a1 = a1->next;
                    top->a2 = a2->next;
                    top->end = a1->end->next;
                    top->s1 = s1_d;
                    top->s2 = s2_d;
                    top++;
                }
                else
                    return(FALSE);
            }
            else if (type_lt(a2) == Var)
            {
                if (occurs(vnum(a2), s2_d, a1, s1_d))
                    return(FALSE);
                else
                {
                    bind(vnum(a2), s2_d, t1, a1, s1_d);
                }
            }
            else
                return(FALSE);
        }
        else if (type_lt(a1) == Var)
        {
            if ((type_lt(a2) == Var) && (a1->val == a2->val) && s1_d == s2_d)
                ;
            else if (occurs(vnum(a1), s1_d, a2, s2_d))
                return(FALSE);
            else
            {
                bind(vnum(a1), s1_d, t1, a2, s2_d);
            }
        }
        else
        {
            if (type_lt(a2) == Constant)
            {

```

```

        if (a1->val != a2->val)
            return (FALSE);
    }
    else if (type_lt(a2) == Var)
    {
        bind(vnum(a2), s2_d, t1, a1, s1_d);
    }
    else
        return (FALSE);
}
a1 = a1_n;
a2 = a2_n;
}
return (TRUE);
}

```

The remaining operation is the application of a substitution to a literal/term to produce a new literal/term. This code includes a few subtle aspects, as well.

1. Consider the case in which a hyperresolvent is constructed by unifying the negative literals in the nucleus with literals from positive clauses. To be specific, suppose that three negative literals exist in the nucleus. Then, the unification operations include four distinct binding environments (substitutions), one for each of the four parent clauses. Any of the substitutions may end up dereferencing to a variable represented in any of the other substitutions. The difficulty comes from trying to keep unbound variables in the different clauses distinct. In particular, the routine to apply the substitution represented by the set of binding environments must know about all of the distinct SUBSTITUTIONs.
2. If a binding was produced by a match operation, the *image* field in a substitution entry will be filled in, but the *env* field will be NULL.

The actual code to apply a substitution (represented by a set of SUBSTITUTIONs) is as follows:

```

/* apply a substitution (represented by a NULL terminated array
   of SUBSTITUTIONs) to a literal/term represented by (term, sub)
*/
LT apply_sub(term, sub, sublist)
LT term;
SUBSTITUTION sub, sublist[];
{
    int i;
    LT apply_bod();

    if (sublist[1] == NULL) /* special case for match -- no env
                           for 1st term */
        sublist[0]->offset = MAXVAR;
    else
    {
        for (i=0; sublist[i] != NULL; ++i)
            sublist[i]->offset = i * MAXVAR;
    }
    return (apply_bod(term, sub));
}

LT apply_bod(term, sub)
LT term;
SUBSTITUTION sub;
{

```

```
LT apply_comp();

Deref(term, sub);
if (type_lt(term) == Var)
{
    if (sub)
        return(make_var((vnum(term)+sub->offset)));
    else
        return(make_var((vnum(term))));
}
else if (type_lt(term) == Constant)
    return(make_constant(symbol_id(term)));
else
    return(apply_comp(term, sub));
}

LT apply_comp(term, sub)
LT term;
SUBSTITUTION sub;
{
    TPOS p1;
    LT a1, a2;
    int i;
    LT args[MAXVAR];
    SUBSTITUTION sub1;

    for (i=0, a1=first_arg(term, &p1);
        a1;
        a1 = next_arg(&p1))
    {
        sub1 = sub;
        Deref(a1, sub1);
        if (type_lt(a1) == Var)
        {
            if (sub1)
                a2 = make_var((vnum(a1)+sub1->offset));
            else
                a2 = make_var(vnum(a1));
        }
        else if (type_lt(a1) == Constant)
            a2 = make_constant(symbol_id(a1));
        else
            a2 = apply_comp(a1, sub1);

        args[i++] = a2;
    }
    args[i] = NULL;
    return(make_complex(functor(term), args));
}
```

This completes our discussion of the algorithms to match two literals/terms, unify two literals/terms, and apply substitutions. Again, we suggest that you study the Appendix to see the exact interface, which is intended to hide the higher-level implementation from the complexities faced at this level.

Exercises:

3. Write a short program *test_unify* that reads two terms and calls *unify* to compute a most general common instance. Have the program compute the unification for 1000 times, and then compare the performance of the simple and optimized version of the unification algorithm.
4. The way that *bind* was coded was fairly inefficient. The code can be improved (in terms of performance) substantially. However, there is little point to doing that, if it won't matter in any event. First, use *gprof* (a UNIX utility) to find out what percentage of unification time is spent in *bind*. You can do this by executing the following steps:

```
gcc -pg test_unify.c fdb.c
a.out < input_terms
gprof > gprof.out
```

where the *gcc* command produces an executable module in *a.out* that records performance statistics. Running *a.out* produces a file that is used as input by *gprof* to produce a detailed report on where the time is going. Read the UNIX manual pages on *gprof*, and study the report. If *bind* is consuming a significant percentage of time, rewrite it as a macro (and temporary variables to avoid all of the duplicated recomputation of pointers), and see how far you can reduce the cost.

2.3. Indexing Literals and Terms

There are three basic operations collectively referred to as "indexing" operations:

1. Given a literal/term *t*, extract from a large set of other literal/terms exactly those that unify with *t*.
2. Given a literal/term *t*, extract from a large set of other literal/terms exactly those that are instances of *t*.
3. Given a literal/term *t*, extract from a large set of other literal/terms exactly those that are generalizations of *t*.

In each case, *t* will actually be a 2-tuple representing a literal/term with possibly bound variables, and the output will be a set of literals/terms with the corresponding substitutions.

A good implementation of these three basic operations is critical to achieving real performance. It is also one of the more difficult tasks in constructing a system. Our approach is to define a "simple" implementation of the operations; this establishes the exact interface and functionality. Later, after building a simple theorem prover that illustrates the use of the formula database, we will cover in detail how to construct a high-performance version of the routines.

For each of the three operations, a given term (represented as a 2-tuple, since variables in the term may be bound) is used to extract a set of "answers." One could have a single request construct the entire list and return it; however, this can be inconvenient when potentially very large numbers of literals/terms may be included in the answer. For, example, if one considers what a request with a given literal of

$$p(V0, V1, V2)$$

might produce (potentially thousands of unifiable literals), it is clear that some other mechanism is required. We choose to use "first/next" calls. That is, a distinct routine is used to acquire the first element of the set of answers, and subsequent calls to a "next" routine are used to get back the other answers. The "first" routine establishes a position in the set of answers, and that position is updated for each subsequent request for the "next" element of the answer set.

The set of literals/terms from which answers are extracted is referred to as an *index*. To understand how an index is used, let us consider how one might implement hyperresolution. We will construct two indexes. One will contain entries for negative literals (that occur in nucleus clauses) and a second will

contain positive literals that occur in positive clauses that are "clashable" (but not positive literals that occur in mixed clauses). Then, given a negative literal from a nucleus, we could access the index of positive, clashable literals to find occurrences of literals that unify with the given literal. Similarly, given a literal from a positive clause, we could use the index of negative literals to find those that unify.

To understand exactly how we intend the indexing operations to be defined, we will use a very simple implementation first. In this implementation, an empty index is initialized by using

```
typedef struct index INDEX;

struct index {
    struct index_nd *link;
};

/* this structure describes a single occurrence of a literal/term
   in a clause.
*/
struct where {
    CLAUSE cl;      /* clause containing the occurrence */
    int sign_lit;   /* 0 -> negative; 1 -> positive */
    int lsub;       /* which literal in the set (either pos or neg) */
    LT term;        /* a pointer to the actual literal/term in the
                     designated literal (i.e., this can point to
                     the LT for the entire literal or to any
                     subterm of the literal)
                     */
};

struct index_nd {
    struct index_nd *link;
    struct where occ;
};

/* initialize an index to empty */
init_index(i)
INDEX *i;
{
    i->link = NULL;
}
```

Occurrences of either literals or terms can be added to an index and deleted from an index by using the following routines:

```
add_to_index(index, occurrence) /* **** simple version *** */
INDEX *index;
struct where *occurrence;
{
    struct index_nd *i;

    if ((i = (struct index_nd *) malloc(sizeof(struct index_nd))) == NULL)
    {
        printf("malloc error in add_to_index0");
        exit(1);
    }
    i->occ = *occurrence;
    i->link = index->link;
    index->link = i;
}
```

```

delete_from_index(index, occurrence)
INDEX *index;
struct where *occurrence;
{
    struct index_nd **p, *p1;

    p = &(index->link);
    while ((*p) && (! ((*p)->occ.cl == occurrence->cl) &&
        ((*p)->occ.sign_lit == occurrence->sign_lit) &&
        ((*p)->occ.lsub == occurrence->lsub) &&
        ((*p)->occ.term == occurrence->term))))
        p = &((*p)->link);

    if (*p)
    {
        p1 = *p;
        *p = p1->link;
        free(p1);
    }
}

```

Once an index has been constructed, one can get the entries that unify with a given literal/term by using the following two routines:

```

typedef struct upos          UPOS;

/* This structure retains a "position" in the set of answers */
struct upos {
    LT term;
    struct substitution *sub1, *sub2;
    struct trail *tr;
    struct index_nd *last;
};

/* Returns TRUE iff a unifiable entry in the index is found.
   In that case, p1 is set to be used to continue looking for
   the remaining answers.
*/
BOOL first_unifiable(index, term, sub1, sub2, occurrence, p1, tr)
INDEX *index;
LT term;
SUBSTITUTION sub1, sub2;
struct where *occurrence;
UPOS *p1;
TRAIL *tr;
{
    struct index_nd *i1;

    for (i1=index->link; i1; i1=i1->link)
    {
        if (unify(term, sub1, i1->occ.term, sub2, tr))
        {
            p1->last = i1;
            p1->term = term;
            p1->sub1 = sub1;
            p1->sub2 = sub2;
            p1->tr = tr;
            *occurrence = i1->occ;

```



```
        return(TRUE);
    }
    return(FALSE);
}

/* Return the next unifiable entry in the index */
BOOL next_unifiable(occurrence,p1)
struct where *occurrence;
UPOS *p1;
{
    struct index_nd *i1;

    undo_bindings(p1->tr);
    for (i1=p1->last->link; i1; i1=i1->link)
    {
        if (unify(p1->term,p1->sub1,i1->occ.term,p1->sub2,p1->tr))
        {
            p1->last = i1;
            *occurrence = i1->occ;
            return(TRUE);
        }
    }
    return(FALSE);
}
```

The routines for first/next instances and generalizations are similar:

```
typedef struct upos          IPOS;

BOOL first_instance(index,term,sub,occurrence,p1,tr)
INDEX *index;
LT term;
SUBSTITUTION sub;
struct where *occurrence;
IPOS *p1;
TRAIL *tr;
{
    struct index_nd *i1;

    for (i1=index->link; i1; i1=i1->link)
    {
        if (match(term,sub,i1->occ.term,tr))
        {
            p1->last = i1;
            p1->term = term;
            p1->sub1 = sub;
            p1->tr = tr;
            *occurrence = i1->occ;
            return(TRUE);
        }
    }
    return(FALSE);
}

BOOL next_instance(occurrence,p1)
struct where *occurrence;
IPOS *p1;
```

```
(
struct index_nd *i1;

    undo_bindings(p1->tr);
    for (i1=p1->last->link; i1; i1=i1->link)
    {
        if (match(p1->term, p1->sub1, i1->occ.term, p1->tr))
        {
            p1->last = i1;
            *occurrence = i1->occ;
            return(TRUE);
        }
    }
    return(FALSE);
)

typedef struct upos          GPOS;

BOOL first_generalization(index, term, sub, occurrence, p1, tr)
INDEX *index;
LT term;
SUBSTITUTION sub;
struct where *occurrence;
GPOS *p1;
TRAIL *tr;
{
    struct index_nd *i1;

    for (i1=index->link; i1; i1=i1->link)
    {
        if (match(i1->occ.term, sub, term, tr))
        {
            p1->last = i1;
            p1->term = term;
            p1->sub1 = sub;
            p1->tr = tr;
            *occurrence = i1->occ;
            return(TRUE);
        }
    }
    return(FALSE);
}

BOOL next_generalization(occurrence, p1)
struct where *occurrence;
GPOS *p1;
{
    struct index_nd *i1;

    undo_bindings(p1->tr);
    for (i1=p1->last->link; i1; i1=i1->link)
    {
        if (match(i1->occ.term, p1->sub1, p1->term, p1->tr))
        {
            p1->last = i1;
            *occurrence = i1->occ;
            return(TRUE);
        }
    }
}
```

```
    return (FALSE) ;  
}
```

The *initialize* routine, the routine to add an entry to an index, and these six search routines define the interface between higher-level routines and the indexing routines. We emphasize that the simple versions just presented are used only to define the interface and to allow us to code and execute a simple theorem prover. Once we have such a program running, we will return to indexing and present a high-performance version of these routines (preserving exactly the interface defined by these simple routines).

These indexing routines conclude our short introduction to the services offered by FDB (the formula database). We urge you to study the original interface document for this FDB (the working document that was used during the writing of the original versions of these routines), which is included in the Appendix. If you are having a hard time understanding exactly why we have defined these routines and how we intend to use them, we hope to make that clear by describing a simple theorem prover in the next section. That system will implement hyperresolution, the set of support strategy, subsumption, and weighting. Although a basically trivial program, we have successfully proven a surprising number of problems with it—many that have been regarded as "hard" in the literature. In any event, it is a useful vehicle for clarifying the exact role played by the routines that we have been calling our *formula database*.

3. A Simple Theorem Prover

We now present a small theorem prover that is quite simple and yet is powerful enough to support the following features:

- hyperresolution
- forward subsumption
- weighting
- iterative deepening

The theorem prover is extremely easy to use because the input consists of a single list of clauses describing the problem; there is no distinction between axioms, set of support, have-been-given, etc., from the user's perspective. The main procedure for the theorem prover follows:

```
#include "tp.h"  
  
/* This theorem prover takes as input a set of clauses.  
   Each clause is on a single line. Rather than give a  
   grammar, let me just give a few examples from the P formulation  
   of the axioms for a group:  
  
       ((P V0 V1 V2) (P V1 V3 V4) (P V0 V4 V5)) ((P V2 V3 V5))  
       ((P b a c)) ()  
  
       () ((P V0 V1 (f V0 V1)))  
       () ((P V0 e V0))  
  
   That is, the clause is represented as two lists of literals;  
   the first being the negative literals and the second the positive.  
   Variable are of the form Vi. Constants and functors are single  
   characters (other than V).  
  
   This is a pretty awful input format, but it will do for purposes  
   of illustration (and it is easy to parse!).  
*/  
  
main()  
{
```

```
int i;

symtab[0] = 0; /* start with an empty symbol table */

init_index(&neg_clashable_index);
init_index(&pos_sub_index);
init_index(&pos_clashable_index);

read_input();

printf("non_positive input clauses:0);
print_list(input_non_positive_clauses);

printf("positive input clauses:0);
print_list(input_positive_clauses);

for (i=0; i < MAX_WT+1; i++)
    positive_clauses_by_wt[i] = NULL;

split_on_weight(input_positive_clauses);

while (!proof_found && max_wt <= MAX_WT)
{
    max_wt += (int) ((max_wt * 0.1) + 0.9);
    printf("begin processing at max_wt %d0,max_wt);
    proof_found = process_at_given_wt();
}
}
```

The procedure *read_input* is a simple loop that reads a clause, renames the variables in it, determines whether the clause should be placed on the list of positive or non-positive clauses, and then places it on that list. The variables in each clause are renamed because many of our procedures assume that V0 is the first variable in each clause, that V1 is the second variable, etc.

```
read_input()
{
    CLAUSE read_clause();
    CLAUSE cl;

    while (cl = read_clause())
    {
        rename_cl(cl);
        if (num_neg_lits(cl) > 0)
        {
            index_nonpos_cl(cl);
            cl->link = input_non_positive_clauses;
            input_non_positive_clauses = cl;
        }
        else
        {
            index_pos_cl(cl,&pos_sub_index);
            cl->link = input_positive_clauses;
            input_positive_clauses = cl;
        }
    }
}
```

After the input clauses have been read, the main procedure splits the positive clauses based on their weight. The weight of a clause is determined to be the sum of the weights of the individual literals, where the weight of a single literal is the maximum of the weights of the individual subterms. The weight of a subterm is simply the count of the number of symbols in it.

```
split_on_weight(clist)
CLAUSE clist;
{
  CLAUSE cl;
  int i;

  /* move clauses to positive lists (based on clause weight) */
  while (clist)
  {
    cl = clist;
    clist = clist->link;

    i = wt_cl(cl);
    cl->link = positive_clauses_by_wt[i];
    positive_clauses_by_wt[i] = cl;
  }
}

int wt_cl(cl)
CLAUSE cl;
{
  int sum = 0;
  int i;

  for (i=0; i < num_neg_lits(cl); i++)
    sum += wt_lit(nth_neg(cl,i));

  for (i=0; i < num_pos_lits(cl); i++)
    sum += wt_lit(nth_pos(cl,i));

  return((sum < MAX_WT) ? sum : MAX_WT);
}

int wt_lit(lit)
LT lit;
{
  int max;
  int i;
  LT a1;
  TPOS p1;

  if (a1 = first_arg(lit,&p1))
    max = wt_trm(a1);
  else
    return(0);

  while (a1 = next_arg(&p1))
  {
    if ((i = wt_trm(a1)) > max)
      max = i;
  }
  return(max);
}
```

```
int wt_term(term)
LT term/
(
  SPOS p1;
  LT t;
  int i;

  t = first_sym(term, &p1);
  i = 1;
  while (t = next_sym(&p1))
    i++;
  return(i);
)
```

After the input clauses are grouped according to weight, a *while* loop in the main procedure (repeated below) implements our iterative deepening technique. The loop begins with a default value for *max_wt* and attempts to find a proof. If the search fails, it increments *max_wt* by 10% (or at least by 1) and begins a new pass, processing the clauses again at the new weight.

```
while (!proof_found && max_wt <= MAX_WT)
(
  max_wt += (int) ((max_wt * 0.1) + 0.9);
  printf("begin processing at max_wt %d\n", max_wt);
  proof_found = process_at_given_wt();
)
```

The procedure *process_at_given_wt* attempts to find a proof at each *max_wt*, invoking a reset procedure on each pass. Given clauses are always taken from the *positive_clauses_by_wt* list. They are immediately moved to the head of the *clashable_satellites* list and are included in the *pos_clashable_index*, permitting them to act as satellite clauses in subsequent resolutions steps.

```
BOOL process_at_given_wt() {
  CLAUSE clashable_satellites = NULL;
  CLAUSE next_given();
  CLAUSE cl;

  while (!proof_found && (cl = next_given(positive_clauses_by_wt)))
  (
    printf("given clause: ");
    print_cl(cl);

    cl->link = clashable_satellites;
    clashable_satellites = cl;

    index_pos_cl(cl, &pos_clashable_index);
    proof_found = generate_hyper_resolvents(cl);
  )

  if (!proof_found)
    reset_for_next_pass(clashable_satellites);
  return(proof_found);
}
```

```
CLAUSE next_given(positive_clauses_by_wt)
CLAUSE positive_clauses_by_wt[MAX_WT+1];
{
  CLAUSE cl;
  int i;

  for (i=0; (i < MAX_WT+1) &&
        ((cl = positive_clauses_by_wt[i]) == NULL); i++)
    ,

  if (i == MAX_WT+1)
    return(NULL);
  else
  {
    positive_clauses_by_wt[i] = cl->link;
    return(cl);
  }
}

reset_for_next_pass(clashable_satellites)
CLAUSE clashable_satellites;
{
  CLAUSE cl;
  struct where occ_of_literal;
  int i;

  for (cl = clashable_satellites; cl; cl=cl->link)
  {
    /* index the positive literals */
    occ_of_literal.sign_lit = 1;
    occ_of_literal.cl = cl;
    for (i=0; i < num_pos_lits(cl); i++)
    {
      occ_of_literal.lsub = i;
      occ_of_literal.term = nth_pos(cl,i);
      delete_from_index(&pos_clashable_index,&occ_of_literal);
    }
  }
  split_on_weight(clashable_satellites);
}

index_pos_cl(cl,indx)
CLAUSE cl;
INDEX *indx;
{
  struct where occ_of_literal;
  int i;

  occ_of_literal.sign_lit = 1;
  occ_of_literal.cl = cl;
  for (i=0; i < num_pos_lits(cl); i++)
  {
    occ_of_literal.lsub = i;
    occ_of_literal.term = nth_pos(cl,i);
    add_to_index(indx,&occ_of_literal);
  }
}
```

The procedure *generate_hyper_resolvents* generates all possible hyperresolvents from the given clause and the clauses currently on the *clashable_satellites* list. Note that it uses the first/next methodology described earlier to generate the clauses. For each generated clause, it checks to see whether the clause is "empty" representing a contradiction and thus a proof. If not, then the clause is positive (by definition of hyperresolution). It is indexed and is placed on the list of *positive_clauses_by_wt*.

```

BOOL generate_hyper_resolvents(given)
CLAUSE given;
{
  CLAUSE cl;
  CLAUSE first_hyper_resolvent();
  CLAUSE next_hyper_resolvent();
  HPOS hpos; /* position in a set of hyperresolvents */
  int ancestors[MAX_LTS_CL+2];
  int i;

  cl = first_hyper_resolvent(given,&hpos,ancestors);
  while (!proof_found && cl)
  {
    if ((wt_cl(cl) > max_wt)
        || fsubsumed(cl)
        || (rename_cl(cl) > MAXVAR))
    {
      dealloc_cl(cl);
    }
    else
    {
      assign_id(cl);
      printf("Derived from (%d",ancestors[0]);
      for (i=1; ancestors[i]; i++)
        printf(",%d",ancestors[i]);
      printf("): ");
      print_cl(cl);

      if (num_pos_lits(cl) == 0)
      {
        printf("PROOF COMPLETED");
        proof_found = TRUE;
      }
      else
      {
        index_pos_cl(cl,&pos_sub_index);
        i = wt_cl(cl);
        cl->link = positive_clauses_by_wt[i];
        positive_clauses_by_wt[i] = cl;
      }
    }
    if (!proof_found)
      cl = next_hyper_resolvent(&hpos);
  }
  return(proof_found);
}

```

The deductions performed by the theorem prover take place in *first_hyper_resolvent*, *next_hyper_resolvent*, and the procedures that they call. *First_hyper_resolvent* assumes that the given clause is a positive clause. It examines each positive literal in the given clause until it finds one that clashes

with some negative literal in the clashable clauses. The clause containing the negative literal then becomes the nucleus. *Finish_hr* is then invoked to complete the computation of the hyperresolvent by finding additional positive clauses containing literals that clash with the negative literals in the nucleus. As long as negative literals are successfully being clashed from the nucleus, *finish_hr* moves forward through them. However, if one literal fails to clash under the substitution being built, *finish_hr* must backtrack and attempt to clash the previous negative literal in another way. If all the negative literals are successfully clashed away, then *form_hyper_resolvent* is called to build the actual resolvent from the positive literals of the satellites.

```
CLAUSE first_hyper_resolvent (given, hpos, ancestors)
CLAUSE given;
HPOS *hpos;
int ancestors[];
{
  int i, j;
  BOOL found_nuc;
  CLAUSE finish_hr();
  struct where nuc_where;

  hpos->ancestors = ancestors;
  hpos->nsub = alloc_sub();
  hpos->cvec[0].sat_sub = alloc_sub();
  hpos->cvec[0].sat_where.cl = given;
  hpos->cvec[0].sat_where.sign_lit = 1;
  hpos->size_cvec = 1;

  for (found_nuc=FALSE, i=0;
       !found_nuc && (i < num_pos_lits(given));
       i++)
  {
    hpos->cvec[0].sat_where.lsub = i;
    hpos->cvec[0].sat_where.term = nth_pos(given, i);

    if (found_nuc = first_unifiable(&neg_clashable_index,
                                     nth_pos(given, i),
                                     hpos->cvec[0].sat_sub,
                                     hpos->nsub,
                                     &nuc_where,
                                     &(hpos->cvec[0].up),
                                     &(hpos->cvec[0].tr)))
    {
      hpos->cvec[0].nucleus_literal = nuc_where.lsub;
    }
  }
  if (found_nuc)
  {
    hpos->nucleus = nuc_where.cl;
    return(finish_hr(hpos));
  }
  else
  {
    cleanup_hpos(hpos);
    return(NULL);
  }
}

cleanup_hpos(hp)
```

```
HPOS *hp;
{
  int i;

  dealloc_sub (hp->nsub);
  for (i=0; i < hp->size_cvec; i++)
    dealloc_sub (hp->cvec[i].sat_sub);
}

CLAUSE finish_hr (hp)
HPOS *hp;
{
  int i,j,k;
  BOOL found_next_nuc_lt;
  CLAUSE form_hyper_resolvent ();
  CLAUSE hr_backtrack ();
  struct where nuc_where;

  if (hp->size_cvec < num_neg_lits (hp->nucleus))
  {
    i = 0;
    found_next_nuc_lt = FALSE;
    while (!found_next_nuc_lt)
    {
      for (j=0; (j < hp->size_cvec) &&
            (hp->cvec[j].nucleus_literal != i); j++)
        ;
      if (j == hp->size_cvec)
        found_next_nuc_lt = TRUE;
      else
        i++;
    }
    nuc_where.cl = hp->nucleus;
    nuc_where.sign_lit = 0;
    nuc_where.lsub = hp->cvec[j].nucleus_literal = i;
    nuc_where.term = nth_neg (hp->nucleus, i);
    hp->cvec[j].sat_sub = alloc_sub ();
    if (first_unifiable (&pos_clashable_index,
                        nth_neg (hp->nucleus, i),
                        hp->nsub,
                        hp->cvec[j].sat_sub,
                        &(hp->cvec[j].sat_where),
                        &(hp->cvec[j].up),
                        &(hp->cvec[j].tr)))
    {
      hp->size_cvec++;
      return (finish_hr (hp));
    }
    else
    {
      return (hr_backtrack (hp));
    }
  }
  else
    return (form_hyper_resolvent (hp));
}

CLAUSE form_hyper_resolvent (hp)
HPOS *hp;
```

[illegible]

```

for (j=hp->cvec[0].sat_where.lsub+1;
    !success && (j < num_pos_lits(hp->cvec[0].sat_where.cl));
    j++)
{
    hp->cvec[0].sat_where.lsub = j;
    hp->cvec[0].sat_where.term = nth_pos(hp->cvec[0].sat_where.cl, j);

    success = first_unifiable(&neg_clashable_index,
                              nth_pos(hp->cvec[0].sat_where.cl, j),
                              hp->cvec[0].sat_sub,
                              hp->nsub,
                              &nuc_where,
                              &(hp->cvec[0].up)
                              &(hp->cvec[0].tr));
}
}
if (success)
{
    hp->nucleus = nuc_where.cl;
    hp->cvec[0].nucleus_literal = nuc_where.lsub;
    return(finish_hr(hp));
}
else
{
    cleanup_hpos(hp);
    return(NULL);
}
}
else
{
    i = hp->size_cvec - 1;
    success = next_unifiable(&(hp->cvec[i].sat_where),
                             &(hp->cvec[i].up));
    if (success)
    {
        return(finish_hr(hp));
    }
    else
    {
        dealloc_sub(hp->cvec[i].sat_sub);
        hp->size_cvec--;
        return(hr_backtrack(hp));
    }
}
}
}

```

A simple forward subsumption test is performed for each generated hyperresolvent. The procedures *fsubsumed* and *fsub_on_lit* take advantage of the fact that hyper-resolvents are positive; the lower-level procedures do not take advantage of this fact, however, and could be used for full subsumption tests. *Fsubsumed* invokes *fsub_on_lit* to examine each literal in the hyperresolvent attempting to find a more general literal in an existing clause. If a more general literal is found, the clause containing it may subsume the hyperresolvent. In such a case, *Fsub_on_lit* completes the test for subsumption by calling *subsumes* with the substitution formed during the generalization test. *Subsumes* checks to see whether the remaining literals in the "subsuming" clause appear in the "subsumed" clause (under the current substitution) by invoking *map_lit* for each literal.

```
BOOL fsubsumed(cl)
CLAUSE cl;
{
  int i;

  for (i=0; i < num_pos_lits(cl); i++)
  {
    if (fsub_on_lit(nth_pos(cl,i),cl))
      return(TRUE);
  }
  return(FALSE);
}

BOOL fsub_on_lit(lit,cl)
LT lit;
CLAUSE cl;
{
  struct where w;
  GPOS pl;
  SUBSTITUTION sub;
  TRAIL tr;

  sub = alloc_sub();
  if (first_generalization(&pos_sub_index,lit,sub,&w,&pl,&tr))
  {
    /* we do not consider longer clauses subsuming shorter ones */
    if (num_pos_lits(w.cl) <= num_pos_lits(cl))
    {
      if (subsumes(w.cl,cl,sub))
      {
        print_cl(w.cl);
        printf(" subsumes ");
        print_cl(cl);
        printf("0");

        dealloc_sub(sub);
        return(TRUE);
      }
    }
    while (next_generalization(&w,&pl))
    {
      if (num_pos_lits(w.cl) <= num_pos_lits(cl))
      {
        if (subsumes(w.cl,cl,sub))
        {
          print_cl(w.cl);
          printf(" subsumes ");
          print_cl(cl);
          printf("0");

          dealloc_sub(sub);
          return(TRUE);
        }
      }
    }
  }
  dealloc_sub(sub);
}
```

```
    return(FALSE);
}

struct sstack_ent {
    BOOL is_positive; /* we need to keep track of the sign */
    LT slit;          /* source lit - lit to map */
    TRAIL t;
    int nxt_lt;       /* next "image" literal to try to map to */
};

BOOL subsumes(subsumer, subsumed, sub)
CLAUSE subsumer, subsumed;
SUBSTITUTION sub;
{
    struct sstack_ent ss[MAX_LTS_CL-1];
    int ss_nxt, ss_end, i;

    ss_end = 0;

    for (i=0; i < num_pos_lits(subsumer); i++)
    {
        ss[ss_end].slit = nth_pos(subsumer, i);
        ss[ss_end++].is_positive = TRUE;
    }

    for (i=0; i < num_neg_lits(subsumer); i++)
    {
        ss[ss_end].slit = nth_neg(subsumer, i);
        ss[ss_end++].is_positive = FALSE;
    }

    ss_nxt = 0;
    while ((ss_nxt > -1) && (ss_nxt < ss_end))
    {
        ss[ss_nxt].nxt_lt = 0;
        if (map_lit(ss+ss_nxt, subsumed, sub))
            ss_nxt++;
        else
        {
            ss_nxt--;
            while (ss_nxt > -1)
            {
                undo_bindings(&(ss[ss_nxt].t));
                ss[ss_nxt].nxt_lt++;
                if (map_lit(ss+ss_nxt, subsumed, sub))
                {
                    ss_nxt++;
                    break;
                }
            }
            else
                ss_nxt--;
        }
    }
    return(ss_nxt == ss_end);
}

BOOL map_lit(s, cl, sub)
struct sstack_ent *s;
```

```
CLAUSE cl;
SUBSTITUTION sub;
{
    if (s->is_positive)
    {
        while ((s->nxt_lt < num_pos_lits(cl)) &&
            !match(s->slit, sub, nth_pos(cl, s->nxt_lt), &(s->t)))
        {
            s->nxt_lt++;
        }
        if (s->nxt_lt < num_pos_lits(cl))
            return(TRUE);
        else
            return(FALSE);
    }
    else
    {
        while ((s->nxt_lt < num_neg_lits(cl)) &&
            !match(s->slit, sub, nth_neg(cl, s->nxt_lt), &(s->t)))
        {
            s->nxt_lt++;
        }
        if (s->nxt_lt < num_pos_lits(cl))
            return(TRUE);
        else
            return(FALSE);
    }
}
```

This seemingly trivial theorem prover has successfully proven a surprising number of problems, many of which have been regarded as hard in the literature. It represents a good introduction to our implementation techniques and is a good starting point for a more powerful theorem prover supporting equality reasoning through such features as paramodulation and demodulation.

4. Adding Support for Full Subsumption and Demodulation

Adding support for a full subsumption test and demodulation requires the use of new indices. Also, an equality-based feature such as demodulation requires that some method be developed for ordering symbols; we have chosen a method known as lexical recursive path ordering (*LRPO*), which is described below. A new version of *main* initializes the indices, reads the *LRPO* parameter values, and initializes the symbol table with equality ("=") as the first symbol.

```
main()
{
    int i;
    priority_clauses = NULL;
    non_positive_clauses = NULL;
    clashable_satellites = NULL;

    proof_found = NULL;
    max_wt = 1;

    symtab[0] = '=';
    symtab[1] = 0; /* start with equality in the symbol table */
}
```

```
for (i=0; i < MAX_SYMS; i++)
    lrpo_status[i] = 0;

init_index(&neg_clashable_index);
init_index(&neg_sub_index);
init_index(&neg_demod_index);
init_index(&pos_sub_index);
init_index(&pos_clashable_index);
init_index(&pos_demod_index);
init_index(&demod_index);

read_lrpo();
read_clauses();

printf("non_positive input clauses:0);
print_list(non_positive_clauses);

printf("positive input clauses:0);
print_list(priority_clauses);

for (i=0; i < MAX_WT+1; i++)
{
    positive_clauses_by_wt[i].first = NULL;
    positive_clauses_by_wt[i].last = NULL;
}

while (!proof_found && max_wt <= MAX_WT)
{
    max_wt += (int) ((max_wt * 0.1) + 0.9);
    printf("begin processing at max_wt %d0,max_wt);
    proof_found = process_at_given_wt();
}
if (proof_found)
    printf("PROOF FOUND at max_wt=%d0,max_wt);
else
    printf("NO PROOF FOUND up to max_wt=%d0,max_wt);
}
```

In examining the new version of *main*, you may have noticed that we have altered the *positive_clauses_by_wt* list so that clauses can now be added to the tail of the list as well as to the head, thus allowing us to treat the list as either FIFO or LIFO. This version of the theorem prover treats the list as FIFO. We have also added a priority clause list which can be used to cause new demodulators or newly demodulated clauses to be selected as given clauses early. The new version of *next_given* selects clauses from the priority list first.

```
CLAUSE next_given()
{
    CLAUSE cl;
    int i;

    if (cl = priority_clauses)
    {
        priority_clauses = cl->link;
        return(cl);
    }
}
```



```
else
{
    for (i=0; (i < MAX_WT+1) &&
        (positive_clauses_by_wt[i].first == NULL); i++)
    {
        if (i == MAX_WT+1)
            return (NULL);
        else
        {
            return del_pos_by_wt(i);
        }
    }
}
```

We also use the priority list at the beginning of each pass to contain clauses that are heavier than the maximum weight for that pass, so that they will be selected early and will go on to become clashable. This use of the priority list is implemented in the new version of *split_on_weight*.

```
split_on_weight(cl)
CLAUSE cl;
{
    int i;

    /* move clauses to positive lists (based on clause weight) */

    index_pos_cl(cl, &pos_sub_index);
    i = wt_cl(cl);
    if (i > max_wt)
    {
        cl->link = priority_clauses;
        priority_clauses = cl;
    }
    else
    {
        add_pos_by_wt(cl, i);
    }
}
```

Process_at_given_wt now removes the given clause from the subsumption index (so that it will not subsume itself), demodulates it, and then performs a forward subsumption test on the demodulated clause. If the clause is not subsumed, back subsumption is performed with it; then it is made clashable and used to generate hyperresolvents. Note that this is the only place at which the back subsumption test is performed.

```
BOOL process_at_given_wt()
{
    CLAUSE next_given();
    CLAUSE cl;
    int i;

    while (!proof_found && (cl = next_given()))
    {
```

```

remove_from_sub_index(cl);
demod_clause(cl, &demod_index);
if (fsubsumed(cl) || (rename_cl(cl) > MAXVAR))
{
    dealloc_cl(cl);
}
else
{
    printf("given clause: ");
    print_cl(cl);

    backsub(cl);
    make_clashable(cl);
    proof_found = generate_hyper_resolvents(cl);
}
}
if (!proof_found)
    reset_for_next_pass(clashable_satellites);
return(proof_found);
}

```

4.1. Full Subsumption

The new index that supports subsumption is called the *neg_sub_index*. Whenever a clause is kept that is negative or mixed, its negative literals are entered (by *make_clashable*) into the *neg_sub_index*; if the clause is positive, then its positive literals are entered into the *pos_sub_index*. Note that a new clause needs to be entered into only one of the two indices, because whenever the clause plays the role of a potential subsumer of another clause, all literals of the subsumer must be mapped into the literals of the subsumed clause. This means that if the subsumer contains any negative literals, they should be a generalization of some literal in the subsumed clause and can thus be located by *fsubsumed* and *fsub_on_lit* through the *neg_sub_index*. Otherwise, the subsumer will be located through the *pos_sub_index*. *Fsubsumed* is the only subsumption procedure that has changed from version one of the theorem prover. *Make_clashable* and *pull_clashable* were added to put/remove clauses into/from the correct indices.

```

BOOL fsubsumed(cl)
CLAUSE cl;
{
    int i;

    for (i=0; i < num_pos_lits(cl); i++)
    {
        if (fsub_on_lit(nth_pos(cl, i), cl, &pos_sub_index))
            return(TRUE);
    }
    for (i=0; i < num_neg_lits(cl); i++)
    {
        if (fsub_on_lit(nth_neg(cl, i), cl, &neg_sub_index))
            return(TRUE);
    }
    return(FALSE);
}

make_clashable(cl)
CLAUSE cl;
{

```

```

if (num_neg_lits(cl) == 0)
(
    cl->link = clashable_satellites; clashable_satellites = cl;
    index_pos_cl(cl, &pos_clashable_index);
    index_pos_cl(cl, &pos_sub_index);
    index_cls_terms_all(cl, 'i', &pos_demod_index);
    if (should_be_demod(cl))
    (
        printf("becoming demod: "); print_cl(cl);
        add_demod_to_index(cl, &demod_index); back_demod(cl);
    )
)
else
(
    cl->link = non_positive_clauses; non_positive_clauses = cl;
    index_non_pos_cl(cl, &neg_clashable_index);
    index_non_pos_cl(cl, &neg_sub_index);
    index_cls_terms_all(cl, 'i', &neg_demod_index);
)
)

pull_clashable(cl)
CLAUSE cl;
(
    CLAUSE *cp;

    if (num_neg_lits(cl) == 0)
    (
        for (cp = &clashable_satellites; *cp != cl; cp = ((*cp)->link))
        ;
        *cp = (*cp)->link;

        unindex_pos_cl(cl, &pos_sub_index);
        unindex_pos_cl(cl, &pos_clashable_index);
        index_cls_terms_all(cl, 'd', &pos_demod_index);

        if (cl->demodulator == TRUE)
            pull_demod_from_index(cl, &demod_index);
    )
    else
    (
        for (cp = &non_positive_clauses; *cp != cl; cp = ((*cp)->link))
        ;
        *cp = (*cp)->link;

        unindex_non_pos_cl(cl, &neg_sub_index);
        unindex_non_pos_cl(cl, &neg_clashable_index);
        index_cls_terms_all(cl, 'd', &neg_demod_index);
    )
)
)

```

Back subsumption is performed by the procedure *backsub* which receives the subsuming clause as a parameter. *Backsub* looks in *neg_sub_index* if the subsuming clause contains any negative literals, and in *pos_sub_index* otherwise. It searches for a clause containing a literal that is an instance of the first literal in the subsumer and then invokes *subsumes* to complete the subsumption test. There may be multiple clauses that are subsumed by the subsumer. Each one is "remembered" until the entire test is completed, and then they are all deleted and pulled from the indices.

```

backsub(cl)
CLAUSE cl;
{
  CLAUSE to_delete[100];
  int td_sub;
  struct where w;
  IPOS p1;
  SUBSTITUTION sub;
  TRAIL tr;
  INDEX *indx;
  LT lit;
  int i;
  BOOL completed = FALSE;

  if (num_neg_lits(cl) > 0)
  {
    lit = nth_neg(cl,0);
    indx = &neg_clashable_index;
  }
  else
  {
    lit = nth_pos(cl,0);
    indx = &pos_clashable_index;
  }

  while (!completed)
  {
    td_sub = 0;
    sub = alloc_sub();
    if (first_instance(indx,lit,sub,&w,&p1,&tr))
    {
      /* we do not consider longer clauses subsuming shorter ones */
      if (num_lits(w.cl) >= num_lits(cl))
      {
        if (subsumes(cl,w.cl,sub))
        {
          to_delete[td_sub++] = w.cl;
        }
      }

      while ((td_sub < 100) && next_instance(&w,&p1))
      {
        if (num_lits(w.cl) >= num_lits(cl))
        {
          if (subsumes(cl,w.cl,sub))
          {
            if (td_sub < 100)
              to_delete[td_sub++] = w.cl;
          }
        }
      }
      dealloc_sub(sub);
    }
    completed = (td_sub < 100);

    while (td_sub-- > 0)
    {
      printf("back subsumed %d with %d0,cl_id(to_delete[td_sub]),cl_id(cl));

```

```
pull_clashable(to_delete(td_sub));  
dealloc_cl(to_delete(td_sub));  
)  
)  
)
```

4.2. Lexical Recursive Path Ordering

Before we discuss demodulation, we should cover the topic of lexical recursive path ordering (LRPO) because LRPO is used in this theorem prover when orienting equality literals, when deciding whether to apply a demodulator, and when deciding whether to make a unit equality a demodulator. We chose LRPO because it is a relatively simple method for making these decisions. It has the side effect that it is supposed to guarantee that demodulation does not loop.

To use LRPO requires that an ordering exist on constant and function symbols. In this theorem prover, the ordering is assumed to be the order in which the symbols appear in the input. This poses no restriction, because the user can supply LRPO information for each symbol in the initial part of the input. Thus, if the user wants the symbol P to have lower ordering than Q, he simply enters the LRPO information for P prior to that for Q. The LRPO information supplied indicates whether the arguments of function symbols are to be regarded as unordered or are to be compared left-to-right or right-to-left. If no information is supplied for a symbol, the default LRPO value is unordered. However, you may supply the LRPO information for a particular symbol, even if you assign it the default value, if you wish that symbol to appear in a particular order for symbols. The left-to-right comparison implies that a functor will be regarded as right associated, and right-to-left implies that it will be regarded as left associated.

The code for our LRPO implementation was borrowed from the automated reasoning program OTTER [4] and altered to fit within this theorem prover. Note that when the code compares two terms, it sometimes compares their arguments. Specifically, it compares them in one of two ways, either as designated by the LRPO flag (i.e., left-to-right or right-to-left) or as multisets (sets in which the arguments are kept in a bag and each subterm appears only once). The major procedure *lpro* takes two terms as arguments that are to be compared and returns TRUE or FALSE indicating whether term 1 > term 2 in the lexicographic recursive path ordering.

```
#define GREATER_THAN 1  
#define LESS_THAN 2  
#define SAME 3  
#define NOT_COMPARABLE 4
```

```
/******
```

```
This file implements lexicographic recursive path ordering which  
is supposed to ensure that demodulation does not loop, but which  
can be expensive in some cases. It is used to orient equalities  
that will be used as demodulators and when applying demodulators.  
In order to compare two terms, you sometimes have to compare their  
arguments. This can be done by examining the set of arguments as  
a multiset (set -> the subterms for each of the term are kept in  
a bag where each subterm appears only once) or by examining them  
one at a time either left to right or right to left (as designated  
by the lr or rl status flags). Note that the lr flag implies that  
the functor is right associated and the rl flag indicates left  
associated (actually it merely implies the order in which subterms  
will be examined).
```

```
*****/
```

```

/*****
*
*   int lrpo(t1, t2) - Is t1 > t2 in the
*   lexicographic recursive path ordering?
*
*****/

int lrpo(t1, t2)
IT t1, t2;
{
    int p, status;
    IT a1;
    TPOS tpos;
    BOOL occurs();

    if (type_lt(t1) == Var)
        /* variable never greater than anything */
        return(0);
    else if (type_lt(t2) == Var)
    {
        /* t1 > variable iff t1 properly contains that variable */
        return((1 << vnum(t2)) & t1->var_occ);
    }
    else if ((symbol_id(t1) == symbol_id(t2)) &&
        (status = lrpo_status[symbol_id(t1)]))
    {
        return(lrpo_lex(t1, t2, status));
    }
    else
    {
        p = sym_precedence(symbol_id(t1), symbol_id(t2));
        if (p == SAME)
            return(lrpo_multiset(t1, t2));
        else if (p == GREATER_THAN)
        {
            /* return (t1 > each arg of t2) */
            if (type_lt(t2) == Complex)
            {
                for (a1=first_arg(t2, &tpos);
                    a1 != NULL && lrpo(t1, a1);
                    a1=next_arg(&tpos))
                {
                    ; /* empty body */
                }
                return(a1 == NULL);
            }
            return(1);
        }
        else /* LESS_THAN or NOT_COMPARABLE */
        {
            /* return (there is an arg of t1 s.t. arg == t2 or arg > t2) */
            if (type_lt(t1) == Complex)
            {
                for (a1=first_arg(t1, &tpos); a1 != NULL; a1=next_arg(&tpos))
                {
                    if (eq_trm(a1, t2) || lrpo(a1, t2))
                        return(1);
                }
            }
        }
    }
}

```

```
        return(0);
    }
} /* lrpo */

/*****
 *
 *   int lrpo_lex(t1, t2, status) -- Is t1 > t2 ?
 *   t1 and t2 have same functor and the functor has lex status.
 *
 *****/

int lrpo_lex(t1,t2,status)
LT t1,t2;
int status;
{
    int i,rc,numargs,a1_idx,a2_idx;
    LT a1,args1[50],a2,args2[50];
    TPOS tpos1, tpos2;

    numargs = arity(t1);
    args1[numargs] = args2[numargs] = NULL;
    if (status == LRPO_RL_STATUS)
        i = numargs - 1;
    else
        i = 0;
    for (a1=first_arg(t1,&tpos1), a2=first_arg(t2,&tpos2);
        a1;
        a1=next_arg(&tpos1), a2=next_arg(&tpos2))
    {
        args1[i] = a1;
        args2[i] = a2;
        if (status == LRPO_RL_STATUS)
            i--;
        else
            i++;
    }

    /* First skip over any identical arguments. */
    /* (Same number of args, because same functor.) */

    for (i=0; i < numargs && eq_trm(args1[i],args2[i]); i++)
        ;

    if (i == numargs)
        rc = 0; /* t1 and t2 identical */
    else if (lrpo(args1[i],args2[i]))
    {
        /* return (t1 > each remaining arg of t2) */
        for (a2_idx=i+1, a2=args2[a2_idx];
            a2 != NULL && lrpo(t1,a2);
            a2_idx++, a2=args2[a2_idx])
        {
            ; /* empty body */
        }
        rc = (a2 == NULL);
    }
    else
    {

```

```
/* return (there is a remaining arg of t1 s.t. arg == t2 or arg > t2) */
rc = 0;
for (a1_idx=i+1, a1=args1[a1_idx];
    a1 != NULL && rc == 0;
    a1_idx++, a1=args1[a1_idx])
{
    if (eq_trm(a1,t2) || lrpo(a1,t2))
        rc = 1;
}

return(rc);

} /* lrpo_lex */

/*****
 *
 * int lrpo_multiset(t1, t2) -- Is t1 > t2 in the lrpo multiset
 * ordering? t1 and t2 have functors with the same precedence.
 *
 *****/

int lrpo_multiset(t1,t2)
LT t1, t2;
{
    LT s1, s2, set_multiset_diff();
    LT a1, a2;
    TPOS tpos1, tpos2;
    int ok;

    s1 = set_multiset_diff(t1,t2); /* builds and returns a set */
    s2 = set_multiset_diff(t2,t1); /* builds and returns a set */

    /* return (s2 not empty and foreach arg a2 of s2
       there is an arg a1 of s1 such that lrpo(a1, a2)).
    */

    a2 = first_arg(s2,&tpos2);
    if (a2 == NULL)
        ok = 0;
    else
    {
        for (ok=1; a2 != NULL && ok; a2=next_arg(&tpos2))
        {
            for (a1=first_arg(s1,&tpos1),ok=0;
                a1 != NULL && ok == 0;
                a1=next_arg(&tpos1))
            {
                ok = lrpo(a1,a2);
            }
        }
    }

    dealloc_lt(s1);
    dealloc_lt(s2);

    return(ok);

} /* lrpo_multiset */
```



```

/*****
 *
 *   LT set_multiset_diff(t1, t2)
 *   Construct the multiset difference, then return the set of that.
 *
 *****/

LT set_multiset_diff(t1,t2)
LT t1,t2;
{
  LT t_result, alloc_lt();
  LT curr, args[50], a1, a;
  TPOS tpos1, tpos;
  int i;

  i = 0;
  args[0] = NULL;
  for (a1=first_arg(t1,&tpos1); a1 != NULL; a1=next_arg(&tpos1))
  {
    /* first check if a preceding occurrence of
       this arg has already been done
    */
    for (a=first_arg(t1,&tpos);
         a != a1 && !eq_trm(a,a1);
         a=next_arg(&tpos))
    {
      ; /* empty body */
    }
    if (a == a1 && num_occurrences(a1,t1) > num_occurrences(a1,t2))
    {
      args[i++] = copy_lt(a);
      args[i] = NULL;
    }
  }

  /* dummy functor */
  t_result = make_complex(MAX_SYMS+1,args);

  return(t_result);
} /* set_multiset_diff */

/*****
 *
 *   num_occurrences(t_arg,t) -- How many times does t_arg occur
 *   as an argument of t?
 *
 *****/

num_occurrences(t_arg,t)
LT t_arg, t;
{
  LT a;
  TPOS tpos;
  int i;

  for (i=0, a=first_arg(t,&tpos); a != NULL; a=next_arg(&tpos))
    if (eq_trm(a,t_arg))
      i++;
}
```

```
        return(i);

    } /* num_occurrences */

    /*****
    *
    *   sym_precedence(sym_num_1, sym_num_2)
    *   Return SAME, GREATER_THAN, LESS_THAN, or NOT_COMPARABLE.
    *
    *****/

    sym_precedence(sym_num_1, sym_num_2)
    int sym_num_1, sym_num_2;
    {
        int p1, p2;

        p1 = sym_num_1;
        p2 = sym_num_2;

        /**** All syms are currently comparable in this tp
        if (p1 == MAX_INT || p2 == MAX_INT)
            return(NOT_COMPARABLE);
        *****/
        if (p1 == p2)
            return(SAME);
        else if (p1 > p2)
            return(GREATER_THAN);
        else
            return(LESS_THAN);
    } /* sym_precedence */
```

4.3. Demodulation

In the definition of *make_clashable* and *pull_clashable* you should have noticed that clauses are inserted into demodulation indices as well as subsumption indices. The new indices added for demodulation are

```
demod_index
neg_demod_index
pos_demod_index
```

Demod_index contains left-hand sides of demodulators. A demodulator is a unit equality literal, and the left-hand side of an equality is simply the first argument of that literal. Given a new clause, if you wish to discover whether any of its terms can be rewritten by an existing demodulator (forward demodulation), *demod_index* can be used to quickly locate left-hand sides of such demodulators. In the reverse situation, if you have a new demodulator and wish to find out whether it can be used to rewrite any terms in existing clauses (backward demodulation), *neg_demod_index* and *pos_demod_index* contain pointers to terms in negative and positive literals that may be rewritten by a demodulator.

Entries are placed into *demod_index* when the decision is made that a clause is to become a demodulator. These entries are created in the procedure *add_demod_to_index*.

```
add_demod_to_index(c,dindex)
CLAUSE c;
INDEX *dindex;
{
    c->demodulator = TRUE;
    index_demod(c,'i',dindex); /* insert demodulator into demod index */
}

pull_demod_from_index(c,dindex)
CLAUSE c;
INDEX *dindex;
{
    index_demod(c,'d',dindex);
    c->demodulator = FALSE;
}

index_demod(c,i_or_d,dindex)
CLAUSE c;
char i_or_d;
INDEX *dindex;
{
    LT lt;
    TPOS tpos;
    struct where occ;

    lt = nth_pos(c, 0);
    lt = first_arg(lt, &tpos);

    occ.cl = c;
    occ.sign_lit = 1;
    occ.lsub = 0;
    occ.term = lt;

    if (i_or_d == 'i')
        add_to_index(dindex, &occ);
    else
        delete_from_index(dindex, &occ);
}
```

All of the terms in a clause are entered into *neg_demod_index* and *pos_demod_index* by the procedure *index_cls_terms_all*.

```
index_cls_terms_all(c,i_or_d,x)
CLAUSE c;
char i_or_d;
INDEX *x;
{
    int i;
    struct where occ;
    LT t1,end1;
```

```
occ.cl = c;
occ.sign_lit = 0;
for (i=0; i < num_neg_lits(c); i++)
{
    occ.lsub = i;
    t1 = nth_neg(c,i);
    endl=t1->end->next;
    for (t1=t1->next; t1 != endl; t1=t1->next)
    {
        if (type_lt(t1) != Var)
        {
            occ.term = t1;
            if (i_or_d == 'i')
                add_to_index(x,&occ);
            else
                delete_from_index(x,&occ);
        }
    }
}

occ.sign_lit = 1;
for (i=0; i < num_pos_lits(c); i++)
{
    occ.lsub = i;
    t1 = nth_pos(c,i);
    endl=t1->end->next;
    for (t1=t1->next; t1 != endl; t1=t1->next)
    {
        if (type_lt(t1) != Var)
        {
            occ.term = t1;
            if (i_or_d == 'i')
                add_to_index(x,&occ);
            else
                delete_from_index(x,&occ);
        }
    }
}
```

There is a set of small procedures that do not actually perform demodulation, but that support it. For example, the procedure *should_be_demod* examines a clause and decides whether it should become a demodulator; it depends on the procedure *is_equality_unit* to determine whether the clause in question is an equality unit clause. Note that it also depends on the LRPO procedures discussed earlier to determine whether the left-hand side is "larger" than the right-hand side of the equality.

```
BOOL should_be_demod(cl)
CLAUSE cl;
{
    TPOS tpos;
    LT lt,f,s;

    if (is_equality_unit(cl))
    {
        lt = nth_pos(cl,0);
```

```
        f = first_arg(lt, &tpos);
        s = next_arg(&tpos);
        return(lrpo(f, s));
    }
    else
        return(FALSE);
}

BOOL is_equality_unit(c)
CLAUSE c;
{
    if ((num_neg_lits(c) == 0) &&
        (num_pos_lits(c) == 1) &&
        Eq(nth_pos(c, 0)))
    {
        return(TRUE);
    }
    else
        return(FALSE);
}
```

Every clause has all of its equality literals reordered such that the left-hand side is heaviest (if possible). Then, if the clause is a unit equality, it is examined to determine whether it should become a demodulator. The procedure *order_equality_lits_in_cl* is responsible for examining each literal in each clause to determine whether it is an equality and whether it needs to be reordered. The procedure *should_reorder* makes the decision to reorder. If reordering is necessary, the work is performed by the procedure *reorder_equality_lit*.

```
BOOL order_equality_lits_in_cl(c)
CLAUSE c;
{
    int i;
    LT lt;
    BOOL reordered = FALSE;

    for (i=0; i < num_neg_lits(c); i++)
    {
        lt = nth_neg(c, i);
        if (Eq(lt) && should_reorder(lt))
        {
            reorder_equality_lit(lt);
            reordered = TRUE;
        }
    }

    for (i=0; i < num_pos_lits(c); i++)
    {
        lt = nth_pos(c, i);
        if (Eq(lt) && should_reorder(lt))
        {
            reorder_equality_lit(lt);
            reordered = TRUE;
        }
    }
}
```

```
    }
    return(reordered);
}

should_reorder(lt)
LT lt;
{
    TPOS tpos;
    LT f,s;

    f = first_arg(lt,&tpos);
    s = next_arg(&tpos);
    return(lrpo(s,f));
}

reorder_equality_lit(lit)
LT lit; /* assumed to be an equality literal */
{
    LT a1,a2,a1_copy,a2_copy;
    TPOS tpos;

    a1 = first_arg(lit,&tpos);
    a2 = next_arg(&tpos);
    a1_copy = copy_lt(a1);
    a2_copy = copy_lt(a2);
    replace_arg(a1,a2_copy);
    replace_arg(a2,a1_copy);
}
```

The small routines that support indexing of demodulators are *add_demod_to_index* and *pull_demod_from_index*, each of which calls the procedure *index_demod* with a flag indicating whether the demodulator is to be added or deleted from the index.

```
add_demod_to_index(c,dindex)
CLAUSE c;
INDEX *dindex;
{
    c->demodulator = TRUE;
    index_demod(c,'i',dindex); /* insert demodulator into demod index */
}

pull_demod_from_index(c,dindex)
CLAUSE c;
INDEX *dindex;
{
    index_demod(c,'d',dindex);
    c->demodulator = FALSE;
}

index_demod(c,i_or_d,dindex)
CLAUSE c;
char i_or_d;
INDEX *dindex;
{
    LT lt;
```

```
TPOS tpos;
struct where occ;

    lt = nth_pos(c, 0);
    lt = first_arg(lt, &tpos);

    occ.cl = c;
    occ.sign_lit = 1;
    occ.lsub = 0;
    occ.term = lt;

    if (i_or_d == 'i')
        add_to_index(dindex, &occ);
    else
        delete_from_index(dindex, &occ);
}
```

New clauses are demodulated by *demod_clause* (which in turn invokes *demod_literal*) as soon as they are generated in *generate_hyper_resolvents* and also when they are chosen as the next given clause. They must be demodulated again when they become given because given clauses come from either the priority list or the positive clauses by weight list, and we do not back demodulate those; we only back demodulate the clashable list.

```
BOOL demod_literal(c,literal,dindex)
CLAUSE c;
LT literal;
INDEX *dindex;
{
    LT t1,t2;
    SUBSTITUTION sub;
    BOOL alpha;
    LT beta;
    struct where dcl;
    TRAIL tr;
    TPOS tpos;
    GPOS gpos;
    SUBSTITUTION sublist[2];
    int count = 100;
    int rc;
    BOOL changed = FALSE;

    if (!EmptyIndex(dindex))
    {
        for (t1=literal; t1; t1=t1->next)
            t1->demodulated = FALSE;

        sub = alloc_sub();
        sublist[0] = sub;
        sublist[1] = NULL;

        /* note that we do not demodulate the "literal"; only args */
        /* moving thru the lit rt to left, i.e. inside out */
        t1 = literal->end;
        while ((t1 != literal) && count)
        {
```

```
/* if not var and not already demodulated */
if ((type_lt(t1) != Var) && !(t1->demodulated))
(
    t2 = t1;
    t1 = t1->prev;

    /* now we demodulate the term referenced by t2 */
    beta = NULL;
    alpha = first_generalization(dindex, t2, sub, &dcl, &gpos, &tr);

    while (alpha && (beta == NULL))
    {
        beta = nth_pos(dcl.cl, 0);
        beta = first_arg(beta, &tpos);
        beta = next_arg(&tpos);
        beta = apply_sub(beta, sub, sublist);
        if (ro = lrpo(t2, beta))
        {
            replace_arg(t2, beta);
            changed = TRUE;
            count--;
            t1 = t1->next->end;
            undo_bindings(&tr);
        }
        else
        {
            dealloc_lt(beta);
            beta = NULL;
            alpha = next_generalization(&dcl, &gpos);
        }
    }

    /* term has been demodulated */
    if (beta == NULL)
        t1->next->demodulated = TRUE;
}
else
    t1 = t1->prev;
}

if (count == 0)
{
    fprintf(stdout, "*** demod limit exceeded; probable loop0);
    fprintf(stdout, "*** 07 07 0);
    fprintf(stderr, "*** demod limit exceeded; probable loop0);
    fprintf(stderr, "*** 07 07 0);
    exit(99);
}

dealloc_sub(sub);
}
return(changed);
}

BOOL demod_clause(c, dindex)
CLAUSE c;
INDEX *dindex;
{
    int i, n;
```



```
    BOOL changed = FALSE;

    for (i = 0, n = num_pos_lits(c); i < n; i++)
        changed |= demod_literal(c, nth_pos(c, i), dindex);

    for (i = 0, n = num_neg_lits(c); i < n; i++)
        changed |= demod_literal(c, nth_neg(c, i), dindex);
    rename_cl(c);

    return(changed);
}
```

Back demodulation is performed with each new clause that is made clashable. The back demodulation procedure *back_demod* invokes the procedure *clauses_to_back_demod* to collect a list of all clauses that can be demodulated. It then invokes *demod_clause* to demodulate each of those clauses. When building its list of clauses to be back demodulated, *clauses_to_back_demod* may find the same clause several times. It uses the procedure *in_to_demod* to search its list of clauses *to_demod* and verify that each clause entered in the list appears only once.

```
back_demod(cl)
CLAUSE cl;
{
    CLAUSE to_demod[100];
    int td_sub;
    LT lit;
    int i;
    BOOL completed = FALSE;
    CLAUSE new_cl;
    SUBSTITUTION sub;

    while (!completed)
    {
        td_sub = 0;
        sub = alloc_sub();
        td_sub = clauses_to_back_demod(cl, to_demod, 100);
        completed = (td_sub < 100);
        while (td_sub-- > 0)
        {
            new_cl = to_demod[td_sub];
            pull_clashable(new_cl);
            demod_clause(new_cl, &demod_index);
            printf("demodulated %d to : ", cl_id(new_cl));
            print_cl(new_cl);
            printf("\n");

            if ((wt_cl(new_cl) > max_wt) ||
                fsubsumed(new_cl) ||
                (rename_cl(new_cl) > MAXVAR))
            {
                dealloc_cl(new_cl);
            }
            else
            {
                new_cl->link = priority_clauses;
                priority_clauses = new_cl;
            }
        }
    }
}
```

```

    )
  )
}

clauses_to_back_demod(c1,to_demod,max_to_demod)
CLAUSE c1;
CLAUSE to_demod[];
int max_to_demod;
{
  struct where w;
  IPOS p1;
  SUBSTITUTION sub;
  TRAIL tr;
  CLAUSE c1,c2;
  int rc,num_to_demod;
  int c1_ud;
  TPOS tpos;
  LT a1;

  /*****
  Note that this procedure does not actually back demodulate
  any clauses, but merely puts a pointer to each which should
  be demodulated in the array to_demod which will hold pointers
  to at most max_to_demod clauses
  *****/

  a1 = nth_pos(c1,0);
  a1 = first_arg(a1,&tpos);
  sub = alloc_sub();
  num_to_demod = 0;
  rc = first_instance(&pos_demod_index,a1,sub,&w,&p1,&tr);
  while (rc && num_to_demod < max_to_demod)
  {
    c1 = w.c1;
    if (c1->id != c1->id &&
        !in_to_demod(c1,to_demod,num_to_demod))
    {
      to_demod[num_to_demod++] = c1;
    }
    rc = next_instance(&w,&p1);
  }
  rc = first_instance(&neg_demod_index,a1,sub,&w,&p1,&tr);
  while (rc && num_to_demod < max_to_demod)
  {
    c1 = w.c1;
    if (c1->id != c1->id &&
        !in_to_demod(c1,to_demod,num_to_demod))
    {
      to_demod[num_to_demod++] = c1;
    }
    rc = next_instance(&w,&p1);
  }
  return(num_to_demod);
}

in_to_demod(c1,to_demod,num_to_demod)
CLAUSE c1;
CLAUSE to_demod[];

```

```
int num_to_demod;
{
int i, found;

for (i=0, found=FALSE; !found && i < num_to_demod; i++)
{
    if (to_demod[i] == c1)
        found = TRUE;
}
return(found);
}
```

5. A High-Performance Implementation of the Indexing Routines

Any poorly designed aspect of a system can become a bottleneck and ultimately determine the maximum achievable performance. When all obvious bottlenecks are removed, however, the indexing routines generally end up determining the performance of a resolution/paramodulation deduction system. The most successful indexing implementations that we have examined were developed by William McCune in the OTTER deduction system[4] and by Jim Christian in the HIPER system[1]. In this section, we describe our own implementation, which was strongly influenced by both of these implementations, as well as by private correspondence with Dan Nicolai of the Institute for Computers and Informatics, Bucharest, Romania.

In Section 2.3, we presented the "simple" version of the indexing routines, which defined the functionality provided by the indexing routines:

```
init_index(i)
INDEX *i;

add_to_index(i, occ)
INDEX *i;
struct where *occ;

delete_from_index(i, occ)
INDEX *i;
struct where *occ;

BOOL first_unifiable(i, trm, sub1, sub2, occ, p1, tr)
INDEX *i;
LT trm;
SUBSTITUTION sub1, sub2;
struct where *occ;
UPOS *p1;
TRAIL *tr;

BOOL next_unifiable(occ, p1)
UPOS *p1;
TRAIL *tr;

BOOL first_instance(i, trm, sub, occ, p1, tr)
INDEX *i;
LT trm;
SUBSTITUTION sub;
struct where *occ;
IPOS *p1;
TRAIL *tr;

BOOL next_instance(occ, p1)
IPOS *p1;
```

```
TRAIL *tr;

BOOL first_generalization(i, trm, sub, occ, p1, tr)
INDEX *i;
L/T trm;
SUBSTITUTION sub;
struct where *occ;
GPOS *p1;
TRAIL *tr;

BOOL next_generalization(occ, p1)
GPOS *p1;
TRAIL *tr;
```

The data structure used in the simple version of the indexing routines was just a singly linked list. Now we will discuss the data structures and routines that we use to efficiently implement the indexing services.

5.1. The Indexing Data Structure

We begin our discussion with some of the definitions used by Nicolaita.

Definition: The *rigid* part of a literal/term T is formed by replacing all of the occurrences of variables in T with "*".

Definition: The *flexible* part of a literal/term T is the list of the occurrences of variables in T .

To clarify these definitions, we consider the rigid part to be

$$i(i(i(*, *), a), i(*, i(*, *)))$$

and the flexible part to be the sequence

$$V1, V0, V0, V2, V1$$

Definition: The *variant* of a flexible part of a literal/term is formed by renaming the variables from the flexible part to $V0, V1, \dots, Vn$.

Thus, the variant associated with the flexible part of the term above is

$$V0, V1, V1, V2, V0$$

Definition: The *permutation* for a flexible part of a literal/term is a correspondence between the variables that occur in a variant and the variables that occur in the flexible part of the literal/term. Note that, although we use the term "permutation," the concept is really of a mapping from the variables in the variant to the particular names used in the actual literal/term. Sometimes, we use the term loosely and speak of the particular pattern of actual variables in the literal/term as "the permutation of the variables in the variant."

The permutation for the flexible part of the term given above would be

$$\begin{aligned} V0 &\rightarrow V1 \\ V1 &\rightarrow V0 \end{aligned}$$

V2 -> V2

Suppose that we need to provide indexing for the literals/terms that occur in a set of clauses S . The data structure that we create will be a discrimination tree that is used to index the set of rigid parts of literals/terms that occur in S . Associated with each rigid part is a set of variants. Associated with each variant, there will be a set of permutations. Each permutation determines a specific literal/term (the one with the given rigid part, with a pattern of variable occurrences given by the variant, and the actual variable names given by the permutation). Finally, associated with each permutation, there will be a set of occurrences, each of which designates an occurrence of the determined literal/term. To make these notions comprehensible, consider the following set of unit clauses:

```

Clause 1:  P (e (e (V0, e (V1, V0)), e (V1, e (V0, V1))))
Clause 2:  P (e (e (V0, e (V1, V2)), e (V2, e (V0, V1))))
Clause 3:  P (e (e (V0, e (V0, V2)), e (V1, e (V1, V2))))
Clause 4:  P (e (e (V0, e (V1, V0)), e (V0, e (V1, V0))))
Clause 5:  P (e (e (V0, e (V0, V0)), e (V1, e (V0, V1))))

```

The rigid part

$e(*, e(*, *))$

has the following variants associated with it:

```

variant 1:  V0, V1, V0
variant 2:  V0, V1, V2
variant 3:  V0, V0, V1
variant 4:  V0, V0, V0

```

Variant 1 has the following permutations associated with it:

```

permutation 1:  V0 -> V0, V1 -> V1
permutation 2:  V0 -> V1, V1 -> V0

```

Permutation 1 has three occurrences associated with it (one in clause 1, and two in clause 4).

The basic technique for computing the set of literals/terms that unify with a given literal/term G can now be given:

1. Compute the rigid part of G . Use a discrimination tree to determine the set of rigid parts that might correspond to terms that unify with G .
2. For each rigid part, look at the set of associated variants. Each variant may be thought of as a term (just plug the variables into the rigid part). Compute the set of variants that unify with G (saving the substitutions).
3. For each successfully unified variant, look at the set of permutations. For each permutation, create a substitution formed by relabeling the substitution elements (i.e., there is no need to use the unification algorithm to compute the unifying substitution; you can compute it directly from the substitution produced by the unification with the variant).
4. Now, for each permutation, gather the associated set of occurrences.

The algorithm that we use bears little resemblance to the steps above. Indeed, you are probably familiar with our approach of "maintaining a position in the computation of the set", passing elements of the set back to the calling routine one-at-a-time.

A little thought should make it clear that the notions of "variant" and "permutation" might substantially reduce the number of required unification operations. In many cases, it will not, and it is not clear at this point that the extra bookkeeping required to maintain the elaborate data structure required to support this type of search is justified. The reason is that, although the notions of "variant" and "permutation" might substantially reduce the number of required unification operations in certain cases, in many cases these notions do not; and the extra bookkeeping required to maintain the elaborate data structure to support this type of search does not seem justified.

We will not go into the details of the code that we employ to implement the above ideas: our version has grown quite complex as a result of attempts to optimize it. You are certainly welcome to study and modify the code that we distribute with FDB, or (even better) produce a comparable set of routines to implement the functionality defined by the simple interface. The specific details are far less important than the general ideas:

1. We recommend the use of a "discrimination tree" (a structure that we called a "dtree" in [3]) to index data structures associated with the rigid part of a literal/term. This data structure is just a standard *trie* (see [2] for an excellent discussion of tries), in which the key is the rigid part of the literal/term. The symbols in the key that are not "*" must each have an associated arity.
2. One has to associate with each leaf of the tree the set of occurrences of literals/terms that have the corresponding rigid part. In our case, we chose to partition these sets. All terms that had the same variant were grouped into a set (each set corresponds to a distinct "variant"). Within each of these subsets, we formed subgroups of occurrences that correspond to the same literal/term (i.e., the occurrences for the same "permutation" of the variables in the "variant").
3. One can use a structure like this to efficiently implement the three fundamental operations of "find the occurrences of terms that unify with this term," "find the occurrences of terms that are instances of this term," and "find the occurrences of terms that are generalizations of this term."
4. The operation of locating generalizations of a given term should almost certainly be implemented by using such a structure. It is not so clear that the other two operations should be implemented this way, however. In particular, consider a situation in which there are many constant symbols (a situation that does arise in applications such as deductive databases). Further, suppose that you need to locate all occurrences of terms that unify with $P(V0,V1,m1323,V2,V3,a1627)$. In this case, other approaches might well perform better.

If you decide to experiment with the actual FDB code, note that debugging can be done using the "simple" version of the routines by just using

```
gcc -DSIMPLE -c -g -O fdb.c
```

when compiling FDB (and a similar command for compiling the routines that invoke FDB services). Later, you can request the optimized versions by using

```
gcc -DFAST -c -g -O fdb.c
```

Finally, we point out that our package of routines includes a short test program (written by McCune) that can be used to measure the performance of just the indexing routines, and thus to compare the performance of different approaches to indexing. McCune has gathered statistics for the current FDB routines and the OTTER indexing routines.

6. Summary

FDB was created to illustrate the basic techniques that can be used to implement high-performance resolution/paramodulation inference systems. We believe that the basic routines that we have presented can be easily used as the basis of an extended system that implements whatever heuristics you wish. We are using FDB as a starting point to build high-performance parallel theorem-proving systems.

References

1. Jim Christian, "High-Performance Permutative Completion," Technical Report ACT-AI-303-89, MCC, Austin, Texas, August, 1989.
2. G.H. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Massachusetts, 1984.
3. J. McCharen, R. Overbeek, and L. Wos, "Complexity and related enhancements for automated theorem-proving programs," *Computers and Mathematics with Applications*, vol. 2, pp. 1-16, 1976.
4. William W. McCune, "Otter 1.0 Users' Guide," Technical Report ANL-88/44, Argonne National Laboratory, Argonne, Illinois, January, 1989.

-END-

DATE FILMED

11 / 20 / 90