Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-264

Mace4 Reference Manual and Guide

by

William McCune

Mathematics and Computer Science Division

Technical Memorandum No. 264

August 2003

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States Government and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

Contents

Mace4 Reference Manual and Guide

William McCune

Abstract

Mace4 is a program that searches for finite models of first-order formulas. For a given domain size, all instances of the formulas over the domain are constructed. The result is a set of ground clauses with equality. Then, a decision procedure based on ground equational rewriting is applied. If satisfiability is detected, one or more models are printed. Mace4 is a useful complement to first-order theorem provers, with the prover searching for proofs and Mace4 looking for countermodels, and it is useful for work on finite algebras. Mace4 performs better on equational problems than did our previous model-searching program Mace2.

1 Introduction

First-order satisfiability is a difficult problem—it's not even semidecidable. Fortunately, many of the satisfiable statements that arise in practice have finite models, and those are what Mace4 looks for. Most of the applications that drove the development of Mace4 are equational, and that is the type of problem on which Mace4 performs well. So if you need help finding small finite algebras, Mace4 is a good candidate.

Most of our work over the past few decades has been in automated theorem proving, that is, searching for proofs. In 1994 I attended a workshop on spectra of finite quasigroups, and that got me interested in searching for finite models of first-order statements. Using ideas from Mark Stickel, Hantao Zhang, John Slaney, and others, I wrote the first version of Mace and used it to work on several quasigroup problems [2]. The early versions of Mace (up through version 2.*) [4], work quite differently from the one described in this report. The early versions transform the problem into a purely propositional (SAT) problem and then apply a Davis-Putnam-Loveland-Logemann (DPLL) procedure. That method works well in many cases, but if the original problem has clauses with many variables or deeply nested terms, the transformation process blows up, and only very small models can be sought.

For a given domain size, the problematic transformation has two stages (which can be done in either order): (1) flattening the clauses, which, in effect, pre-processes the equality inferences, and (2) generating all of the ground instances over the domain. The first stage, which allows us to use a purely propositional decision procedure (without equality), causes the blowup.

If we skip the first stage, we get a set of ground clauses with equality, which is clearly a decidable problem. Jian Zhang's FALCON [7] works in this way, with a special-purpose decision procedure. In particular, it has an optimization, the *least-number heuristic* (LNH), to take advantage of symmetries in the problem. Shortly after FALCON was developed, Jian Zhang and Hantao

Zhang produced SEM [8], which inherited those key methods of FALCON, and which has been distributed and used in several projects.

Mace4 owes much to FALCON and SEM. It transforms the original problem to a ground clauses with equality and uses a special-purpose decision procedure with the LNH. It also uses negative inference rules as in FALCON and SEM.

2 A Little Motivation

Suppose you wish to see a small noncommutative group. You prepare the following input file, named `group.in`.

```
assign(iterate_up_to, 10).  % search up through order 10

clauses(theory).
E * x = x.                  % left identity
x' * x = E.                 % left inverse
(x * y) * z = x * (y * z).  % associativity
A * B != B * A.             % A and B do not commute
end_of_list.
```

Then, you give the input to Mace4 to look for a model as follows.

```
% mace4 < group.in > group.out
```

Mace4 succeeds, and the output file contains the following noncommutative group of order 6.

```
- Model 1 at 0.01 seconds -

  E : 0    A : 1    B : 2

  ' :    0 1 2 3 4 5
      ---------------
         0 1 2 4 3 5

  * :  | 0 1 2 3 4 5
      --+------------
      0 | 0 1 2 3 4 5
      1 | 1 0 3 2 5 4
      2 | 2 4 0 5 1 3
      3 | 3 5 1 4 0 2
      4 | 4 2 5 0 3 1
      5 | 5 3 4 1 2 0
```

3 The Input File

The input file starts with an optional set of commands that tell Mace4 how to search. The rest of the file contains lists of clauses or formulas representing the theory, that is, the clauses or formulas you wish to model. Comments can appear on any line in the file; they start with the character "%" and end at the end of the line.

3.1 The Commands

Mace4 has only a few types of command. The `set` and `clear` commands apply to flags, the `assign` commands apply to integer-valued parameters, and the `op` commands are syntactic sugar, allowing expressions to be written in more human-friendly ways.

3.1.1 Basic Commands

`assign(domain_size, n).        % default n = 2`

> Search for models of size n. See the following command if you wish for Mace4 to iterate through larger domain sizes. The default value of `domain_size` is $n = 2$.

`assign(iterate_up_to, n).      % default n = 0`

> If $n >$ `domain_size`, Mace4 will iterate through domain sizes, restarting the search each time. For example if `domain_size`=4 and `iterate_up_to`=6, Mace4 will try 4, then 5, then 6, until those sizes are exhausted or until some other limit such as `max_models` applies.

```
set(print_models).              % default set
clear(print_models).
```

> If this flag is set, all found models are printed to the output file. The format of the models depends on the state of the flag `print_models_portable`.

```
set(print_models_portable).
clear(print_models_portable).  % default clear
```

> If this flag is set, models are printed to the output file in *portable format*, which means that they can be read by various other programs, for example, a program that takes a list of models and removes isomorphic ones, or a program that uses models to filter clauses.

`assign(max_models, n).         % default n = 1`

> The search will terminate when the nth model is found. (Typically, many of the models are isomorphic; see the program `isofilter` in Section 8.) If domain-size iteration is occuring, the model count carries over from each size to the next.

```
assign(max_seconds, n).          % default n = ∞
```

The search will terminate if it reaches n seconds (user CPU time on UNIX-like systems). If domain-size iteration is occuring, the timer carries over from each size to the next.

```
assign(max_megs, n).             % default n = 192
```

The search will terminate if it tries to dynamically allocate (malloc) more than n megabytes of memory. (The entire Mace4 process can consume somewhat more than `max_megs` megabytes.)

```
set(prolog_style_variables).
clear(prolog_style_variables). % default clear
```

A rule is needed to distinguish variables from constants in clauses. Normally, variables start with (lower case) `u` – `z`, and all other nullary symbols are constants. If this flag is set, variables start with (upper case) `A` – `Z`.

```
set(verbose).
clear(verbose).                  % default clear
```

If this flag is set, the output file receives information about the search, including the initial partial model (the part of the model that can be determined before backtracking starts) and timing and other statistics for each domain size. (It does not give a trace of the backtracking, so it does not consume a lot of file space.)

```
op(precedence, type, symbol).
op(precedence, type, list-of-symbols).
```

This command allows expressions to be written in infix, postfix, and prefix forms without so many parentheses. For example, the command `op(400,infix,*)` declares "`*`" to be an infix (binary) operation with precedence 400. This command affects how clauses and formulas are parsed and printed only. The accepted types for binary operations are `infix`, `infix_left` (left association), and `infix_right` (right association). The accepted types for unary operations are `prefix` and `postfix`. The precedence should be in the range 1–998. Symbols with lower precedence bind more tightly. Several operations are predeclared—see the table in Section 3.2.2.

3.1.2 Advanced Commands

The commands in this sections can be used to fine-tune the search.

```
set(lnh).                        % default set
clear(lnh).
```

This flag says to use the *least number heuristic* (LNH). It is nearly always recommended. See Section 5.4.

```
assign(selection_order, n).    % default n = 2
```

This parameter determines the set of cells that are candidates for selection. See Section 5.3. If $n = 0$, all open cells are considered. If $n = 1$, *concentric order* is used. If $n = 2$, *concentric-band order* is used.

```
assign(selection_measure, n).  % default n = 4
```

Given a set of cells, this parameter determines which of those cells is selected for assignment. See Section 5.3. If $n = 0$, select the first candidate cell. If $n = 1$, select the candidate with the greatest number of occurrences in the current set of (ground) clauses. If $n = 2$, select the candidate that would cause the greatest number of propagations. If $n = 3$, select the candidate that would cause the greatest number of contradictions. If $n = 4$, select the candidate with the fewest possible values.

```
set(negprop).                  % default set
clear(negprop).
```

This flag enables or disables the *negative propagation* inference rules, which derive negated equations. These rules are not required for completeness of the search, but they nearly always help. Four types of negative propagation are enabled or disabled by the following four parameters. For any negative propagation to occur, `negprop` must be set.

```
set(neg_assign).               % default set
clear(neg_assign).
```

If this flag is set, negative propagation is triggered by *assignments* (i.e., when a cell is given a value), for example, $f(2, 3) = 4$.

```
set(neg_assign_near).          % default set
clear(neg_assign_near).
```

If this flag is set, negative propagation is triggered by *near assignments*, for example, $f(2, g(3)) = 4$.

```
set(neg_elim).                 % default set
clear(neg_elim).
```

If this flag is set, negative propagation is triggered by *eliminations*, for example, $f(2, 3) \neq 4$.

```
set(neg_elim_near).            % default set
clear(neg_elim_near).
```

If this flag is set, negative propagation is triggered by *near eliminations*, for example, $f(2, g(3)) \neq 4$.

```
set(trace).
clear(trace).                  % default clear
```

If this flag is set, detailed information about the search, including a trace of all assignments and backtracking, is printed to the standard output. *This flag causes a lot of output, so it should be used only on small searches.*

3.2 The Clauses and Formulas

The theory to be modeled can be specified with first-order clauses, first-order formulas, or a combination of the two. There can be any number of clause and formula lists, given as in the following (silly) example.

```
clauses(theory_part_1).   % associativity of * as a clause
(x * y) * z = x * (y * z).
end_of_list.

formulas(theory_part_2).  % commutativity of * as a formula
all x all y (x * y = y * x).
end_of_list.
```

The names of the lists need not be "`theory_part_1`" and "`theory_part_2`". Any names can be used. When formulas are given, they are immediately transformed (including Skolemization) to clauses by a straightforward procedure.

The clause and formula languages accepted by Mace4 are determined by a library of automated deduction routines that is still evolving (LADR [5]), so they are likely to change in future releases of Mace4. Therefore we do not give a full definition here. Instead we list some key points and examples, and we refer the reader to the sample input files that accompany the Mace4 distribution packages. A more formal description of the languages may appear on the Mace4 Web page [6].

Parsing and printing properties of binary and unary symbols can be declared, allowing infix, prefix, and postfix notation. These declarations are syntactic sugar only—they have no logical significance. The user can declare such properties with the `op` command, and several are predeclared as shown in the following table.

```
op(800, infix,        ->).  % implication
op(800, infix,        <-).  % implied by
op(800, infix,       <->).  % equivalence
op(790, infix_right,   |).  % disjunction
op(780, infix_right,   &).  % conjunction
op(300, prefix,        ~).  % negation
op(700, infix,         =).  % equality
op(700, infix,        !=).  % negated equality
op(500, infix,         +).  % general use
op(400, infix,         *).  % general use
op(300, postfix,       ').  % general use
```

Predeclared Operations

Whitespace (spaces,newlines,returns,tabs,vertical-tabs,formfeeds) is allowed just about anywhere. When in doubt about how much white space and how many parentheses to use, be conservative and observe how Mace4 echoes your input. Mace4 frequently prints more whitespace than necessary.

Negated equality can be abbreviated with the symbol "`!=`".

3.2.1 Domain Elements

If the input clauses or formulas contain constants that are natural numbers, $\{0, 1, 2, 3, \ldots\}$, such constants are always interpreted as members of the domain. For example, one can use 1 and 0 as the top and bottom of a lattice, and the corresponding members of the domain are assigned to them.

When using domain elements (i.e., natural number constants) in the input, it is best to use the smallest ones—doing otherwise defeats the purpose of the least-number heuristic. *If the input contains a domain element that is greater than or equal to the specified domain size, a fatal error occurs.*

3.2.2 Clauses

A clause is a disjunction of literals, terminated with a period. A literal is either an atomic formula or the negation of an atomic formula. The disjunction symbol is `|`, and the negation symbol is `~`.

All quantification in clauses is universal and implicit. Variables are distinguished from constants by the following rule. Ordinarily, variables start with (lower case) `u`–`z`. If the flag `prolog_style_variables` is set, variables start with (upper case) `A`–`Z`.

```
p(x) | ~q(a) | r(x,b).          % a basic clause
|(p(x),|(~(q(a)),r(x,b))).      % the same basic clause
x*y!=z | x*w!=z | y=w.          % a cancellation rule
x * (y + z) = x * y + x * z.    % distributivity rule
x' ' = x.                       % whitespace required between primes
```

Examples of Clauses, Assuming the Default Operator Declarations

3.2.3 Formulas

All quantification in formulas is explicit, and the top level of a formula does not have free variables. Therefore, no rules are needed to distinguish variables from constants—a nullary symbol is a variable if and only if it is bound by a quantifier. For example, the symbol `e` can be a variable. To prevent a common error, however, we do not allow constants that might be misinterpreted by the user as variables. For example, the string `(all x p(x,y))` is not accepted as a formula, because the constant `y` looks like a variable.

- An atomic formula is a formula.
- If `F` is a formula, then `(~F)` is a formula.
- If `F` and `G` are formulas, then `(F & G)`, `(F | G)`, `(F -> G)`, `(F <- G)`, and `(F <-> G)` are formulas.
- If `x` is a symbol and `F` is a formula, then `(all x F)` and `(exists x F)` are formulas.

The default operator declarations allow many parentheses to be dropped. Also, a special rule allows parentheses to be dropped for sequences of quantifiers. For example, the following two strings represent the same formula.

```
all x all y exists z (p(x,y,z) <-> q(x) & ~r(y) | s(z)).
(all x (all y (exists z ((p(x,y,z) <-> (q(x) & (~r(y))) | s(z)))))).
```

4 Running Mace4

The basic way to call Mace4 is shown in the following example.

```
% mace4 < group.in > group.out
```

Mace4 also accepts command-line arguments corresponding to the options described in Section 3.1. Command-line options override the corresponding settings in the input file. To see the correspondence between the options and the command-line arguments, run the following command.

```
% mace4 help
```

Assume the command-line argument `-n` corresponds to the parameter `domain_size`, and `-m` corresponds to `max_models`. If the input file contains the command `assign(iterate_up_to,10)`, the command

```
% mace4 -n8 -m20 < group.in > group.out
```

tells Mace4 to search for up to 20 models of sizes 8, 9, and 10.

4.1 Theorem Prover Compatibility Mode

One of the basic ways to use Mace4 is as a complement to a theorem prover, with the prover searching for a proof and Mace4 looking for a counterexample. In order to be able to use the same input file with both programs, Mace4 has a mode in which it will allow (and ignore) commands and lists intended for other programs. (Ordinarily an unrecognized command or list causes a fatal error.)

The command-line option `-c` tells Mace4 to allow unrecognized `set`, `clear`, and `assign` commands and to allow unrecognized lists of objects.

Mace4 is not compatible with Otter [3] in this way. However, a script can help in transforming Otter (and Mace2) input files for use with Mace4 (see Section 8).

4.2 Exit Codes

When a Mace4 process terminates, it returns an exit code. If Mace4 is called from another program, say a shell or Perl script, that program can use Mace4's exit code to decide what do do next. The exit codes are as follows.

0: The specified number of models (`max_models`, default 1) was found.

1: A fatal error occurred. This is usually caused by an error in the input file, the memory limit (`max_megs`, default 192), or a bug in Mace4.

2: The search completed without finding any models. That is, there are no models within the given constraints.

3: Some models were found, but the search completed before `max_models` models were found.

4: Some models were found, but the time limit (`max_seconds`, default ∞) terminated the search before `max_models` models were found.

5: The time limit (`max_seconds`) terminated the search before any models were found.

Here is an example of a Perl program that calls Mace4. It takes a stream of clauses and for each, it calls Mace4 to look for a noncommutative model up through size four. If none is found, the clause is printed.

```
#!/usr/bin/perl -w
$mace4 = "/home/mccune/bin/mace4";
$input = "/tmp/mace$$";
$unsatisfiable_exit = 2;
while ($equation = <STDIN>) {
  open(FH, ">$input") || die "Cannot open file $input";
  print FH "clauses(theory). $equation f(0,1)!=f(1,0). end_of_list.\n";
  close(FH);
  $rc = system("$mace4 -N4 < $input > /dev/null 2> /dev/null");
  $rc = $rc / 256; # This gets the actual exit code.
  if ($rc == $unsatisfiable_exit) { print $equation; }
}
system("/bin/rm $input");
```

5 How Mace4 Works

This section describes how Mace4 works for a fixed domain size, say n. The members of the domain are always named $\{0, \ldots, n-1\}$. If Mace4 is iterating through domain sizes, this section applies, separately, to each domain size.

First, tables for the function and predicate symbols are set up, and all ground instances of the input clauses (over the domain) are generated. Then, in a systematic way, a recursive backtracking procedure fills in the cells of the tables and uses the ground clauses to propagate the effects of the assignments. When contradictions (dead ends) are encountered, backtracking occurs, the propagations and assignments are undone, and other assignments are attempted. If all the tables become full, with no contradictions, a model has been found. That is, we have an interpretation in which all of the input clauses are true.

5.1 Initialization

Mace4 starts by allocating a table for each function and predicate symbol. Constants (which are function symbols of arity 0) are single cells, function symbols of arity 2 get an $n \times n$ table, and so on. The range of values for function symbols will be members of the domain $[0, \ldots, n-1]$, and values for predicate symbols will be 0 and 1, representing the truth values.

Then, for each input clause, all ground instances over the domain are generated. If a clause has v variables, it has n^v instances. A variable *max_constrained* contains the value of the maximum constrained value and is used for the *least number heuristic* (see Section 5.4). If any domain values occur in the input clauses, *max_constrained* is initialized to the greatest domain value in the input; otherwise it is initialized to -1.

In many cases, the input clauses allow us to start filling in the tables before the search begins. For example, if the input says that a binary operation is idempotent, say $f(x, x) = x$, then the n cells on the diagonal of table f can be filled. This can be done by simply calling the assignment and propagation routines described below. All of these are fixed assignments which never need to be undone.

At this point, the initialization is complete, and the initial partial model (possibly empty) may be printed to the output file. Now, we can start the search.

5.2 Search

Here is the recursive backtracking search procedure, roughly stated.

```
procedure search:
{
    cell = select_cell();
    top = last_value_to_consider();
    foreach i (0 ... top)
    {
        ok = assign_and_propagate(cell,i);
        if (ok)
        {
            search();
            undo_assignments();
        }
    }
}
```

The important components of the search are cell selection, determining which values need to be considered for assignment to the selected cell, and propagating the assignments. Note that the same pseudo-code could be used to represent a Davis-Putnam-Loveland-Logemann (DPLL) propositional satisfiability procedure. In that case, `select_cell()` would select a propositional variable, and `top` is always 1.

5.3 Cell Selection

Cell selection is divided into two stages. The first, controlled by the parameter `selection_order`, determines a set of candidate cells. The second stage, controlled by the parameter `selection_measure`, picks one of the candidates cells.

Candidates Cells. Three methods are available, specified by the parameter `selection_order`.

0. The *linear order*. All open cells are candidates. This order usually defeats the purpose of the least number heuristic.
1. The *concentric order*. Let i be the smallest maximum index of an open cell; then all cells with maximum index i are candidates. This method causes the maximum constrained value to be increased only when necessary.
2. The *concentric-band order*. All cells with maximum index less than or equal to the current maximum constrained value are candidates. If there is none, we revert to concentric order. This method usually gives more candidates than the concentric order, but it still keeps maximum constrained value as low as possible.

Selecting a Candidate. Four methods are available, specified by the parameter `selection_measure`.

0. Select the first candidate.
1. Select the candidate with the greatest number of occurrences in the current set of (ground) clauses. Recall that each cell is associated with a term, e.g., $f(2,3)$. The motivation for this

measure is that the assignment will have the greatest initial effect (ignoring propagation) in simplifying the clauses.

2. Select the candidate that would cause the greatest number of propagations. This is done as follows. For each cell being considered, all assignments and subsequent propagations are done (and undone), and the total number of propagations is counted; this is an expensive lookahead operation with the motivation of filling the tables as soon as possible.

3. Select the candidate that would cause the greatest number of contradictions; for each cell being considered, all assignments and subsequent propagations are done (and undone), and the number of assignments that lead directly to contradictions are counted; this is an expensive lookahead operation with the motivation of cutting off paths as soon as possible.

4. Select the candidate with the smallest number of possible values. Each cell has a list of possible values (see Section 7). As the search progresses, negated equalities can be derived. If, for example, $f(2,3) \neq 4$ is derived, the value 4 is removed from the possible-values list for cell $f(2,3)$. (When all but one value is eliminated, an assignment can be made.) The primary benefit of negative propagation (see Section 5.6) is that it derives negated equalities that are used for this purpose.

5.4 Cell Assignment

Once we select an open cell, we have to determine the set of values to be considered for assignment to the cell. If it is a Boolean-valued cell, we always try both 0 and 1. Otherwise, if the `LNH` (least-number heuristic) flag is clear, we try all members of the domain $[0, \dots, n-1]$.

The *least number heuristic* (LNH) was first used in Falcon [7] and later in SEM [8]. It eliminates some of the isomorphism in the search. The basic idea is that at a given point in the search, all of the domain values, say $[0, \dots, n-1]$, are partitioned into the constrained values $[0, \dots, i]$ and the unconstrained values $[i+1, \dots, n-1]$. The unconstrained values are all symmetric—if any one of them is tried, the others need not be. We try the least of them. That is, when faced with an empty cell, we try $[0, \dots, i+1]$. A value becomes constrained when it is assigned to a cell or when it becomes the index of a cell that is assigned to. (Subsequent assignments during unit propagation do not constrain values. Because of this distinction, we often think of two types of assignment: by selection and by propagation.) The LNH eliminates only some of the isomorphism in the search. In practice, it usually has effect only at the very top levels of the search tree.

5.5 Positive Propagation

When we make an assignment, say $f(2,3) = 4$, positive propagation is always applied. All occurrences of $f(2,3)$ in the ground clauses are rewritten to 4, and these changes can trigger other rewrite operations by the other current assignments. During this process, more assignments can be derived. For example, with equation $g(f(2,3)) = 5$, we rewrite once, and then we have assignment $g(4) = 5$ to propagate. And so on. Positive propagation is important to the efficiency of the method; there are typically many assignments by propagation for each assignment by selection. Each rewriting operation and each assignment are recorded on a stack so that they can be undone when backtracking. In addition, we maintain a queue of assignments to be made and propagated.

5.6 Negative Propagation

Negative propagation derives negated equalities. The power of this feature is in eliminating possible values. If all but one value for a cell is eliminated, an assignment can be made. In addition, cell selection can be based on the least number of possible values, under the motivation that fewer possible values causes less branching.

Suppose we know that $f(2,3) = 4$ and $f(2,g(5)) \neq 4$; then we may infer $g(5) \neq 3$. The same conclusion can be drawn if the signs on the premises are reversed. Examples of the four types of clause to which negative propagation can be applied are shown in the following table.

Type	Example
Assignment	$f(2,3) = 4$
Near assignment	$f(2,g(5)) = 4$
Elimination	$f(2,3) \neq 4$
Near elimination	$f(2,g(5)) \neq 4$

Assignments pair with near eliminations, and eliminations pair with near assignments. Any of the four types can trigger negative inferences, and there are four corresponding flags to enable the negative inferences. For example, if the `neg_elim_near` flag is set, and if a near elimination clause is derived, it will be used with all applicable assignment clauses to derive new negated equalities. Negative inferences can cause propagation of assignments as well as eliminations by eliminating all but one possible value of a cell.

Negative inference and propagation apply to Boolean-valued terms as well as to ordinary terms. If we have $P(3,4)$ and $\neg P(3,g(5))$, we can derive $g(5) \neq 4$. All of the rules for Boolean terms are analogous to the rules for ordinary terms.

Negative propagation is not required for completeness and can be disabled with the command `clear(neg_prop)`.

6 Implementation

Mace4 is coded in C, and and we have tried to make it run quickly. As in most DPLL procedures, the vast majority of the processing time is spent propagating assignments. In our case, that includes both positive and negative propagation.

The data structures were designed with the goal of minimizing the the allocation and deallocation of memory. After the ground clauses are constructed, no more term allocation occurs; all of the rewriting is done by copying pointers. Memory is allocated and deallocated (by our own routines, not by operating system calls) for the stack of changes that must be undone and for the queue of tasks to be done.

Say we have a new assignment $f(1,2) = 3$. We maintain a list of occurrences of $f(1,2)$ (constructed from the term nodes so that it need not be allocated or deallocated); for each occurrence, say $g(f(1,2))$, we rewrite it to $g(3)$ and record that change on the stack. We then update the list of occurrences of $g(3)$ to include this new one and record that change in the stack. If this rewrite creates a new assignment, elimination, near assignment, or near elimination, we add that to the queue of tasks.

For negative inference, we have to be able to quickly locate near assignments and near eliminations as well as ordinary assignments and eliminations. We index near assignments and near eliminations with *complete discrimination trees*, that is, a discrimination tree [1] in which all possible branches are constructed at the start of the search, and only the leaves are updated for insertions and deletions. The complete discrimination trees are not very big, because the term depth is always 2. Insertions into the discrimination trees are recorded on the stack so that they are undone on backtracking.

Mace4 is the first released program that has been constructed with LADR (Library of Automated Deduction Routines) [5], a library of C routines for building automated deduction tools. LADR has been under development at Argonne, off and on, for several years (at times LADR was also known as OPS). The LADR features we used involve the term data structure, parsing and printing terms, interpretations, auxiliary data structures such as linked lists, flag and parameter handling, and clocks.

7 Theory and Completeness

For a given (finite) domain size, the existence of a model is obviously decidable, and Mace4 is intended to be a decision procedure. We can view the existence problem as satisfiability of a set of ground clauses with equality.

To take this view, we start with the set of ground clauses that Mace4 generates during initialization. Then we add some clauses that correspond to operations that are built in to Mace4. First, we add $(n^2 - n)/2$ clauses stating that the n domain members are distinct. Second, for each cell we add an n-literal positive clause stating that the term corresponding to the cell must equal one of the domain members. For example,

$$f(2,3) = 0 \mid f(2,3) = 1 \mid f(2,3) = 2 \mid f(2,3) = 3.$$

Now we have a satisfiability problem for domain size n, independent of Mace4.

Practical decision procedures for ground clauses with equality have been studied before. However, the structure of our problems leads us to believe that the special-purpose methods embodied in Mace4 are better than general-purpose methods that have been proposed. First, the n-literal positive clauses indicate that the splitting and backtracking approach in Mace4 is appropriate. Second, it is not clear how the symmetry exploited by the least-number heuristic could be effectively addressed in a general-purpose method. Third, rewriting with assignment clauses is the only kind of rewriting we need, because we can always get enough assignment clauses (by splitting on the n-literal positive clauses) to rewrite every clause to 0 or 1. Finally, negative inference is not needed for completeness, but it frequently gives enormous speedups.

Before designing Mace4, we discussed the problem with N. Shankar, R. Niewenhuis, and D. Kapur. Most of those discussions revolved around combining congruence closure (to do the ground rewriting) with the DPLL method (to do the splitting and backtracking). That is still an interesting approach, but we took the special-purpose path, because it seems simpler and more natural for this application.

8 Auxiliary Programs

The Mace4 distribution packages contain several additional programs and scripts. Most of these are short C programs built from the routines in LADR library, and they take inputs similar to the Mace4 input.

Isofilter. The least number heuristic in Mace4 prevents some of the isomorphism in the search for models, but it is far from optimal. If the user asks for more than one model for a particular input (e.g., `assign(max_models,10000)`, most of the models will usually be isomorphic to others in the set. The program isofilter takes a stream of models and eliminates the isomorphic ones. Let's say we wish to see all of the ortholattices (OLs) up through order 10. We prepare the input following file, named `OL.in`.

```
op(400, infix, ^).  op(400, infix, v).
assign(iterate_up_to, 10).
set(print_models_portable).
assign(max_models, 100000).

clauses(theory).
% Lattice theory (LT) 6-basis
    x v y = y v x.              x ^ y = y ^ x.
    (x v y) v z = x v (y v z).  (x ^ y) ^ z = x ^ (y ^ z).
    x v (x ^ y) = x.            x ^ (x v y) = x.
% Add the following pair for complemented lattice (CL)
    x v c(x) = 1.
    x ^ c(x) = 0.
% Add the following for ortholattice (OL)
    c(x ^ y) = c(x) v c(y).
    c(x v y) = c(x) ^ c(y).
    c(c(x)) = x.
% Add the following for orthomodular lattice (OML)
% x v (c(x) ^ (x v y)) = x v y.   % OML law
% Add the following for modular ortholattice (MOL)
% x v (y ^ (x v z)) = (x v y) ^ (x v z).  % Modularity
% Add the following for Boolean algebra (BA)
% x v (y ^ z) = (x v y) ^ (x v z).  % Distributivity
end_of_list.
```

Then we run the following command.

```
% mace4 < OL.in | get_interps | isofilter > OL.out
```

The output file `OL.out` contains 24 ortholattices of orders 2–10, and the last line of the file is

```
isofilter: input=1315, kept=24, checks=1291, perms=2975947, 2.27 sec.
```

saying that isofilter received 1,315 interpretations from Mace4 and determined that 24 of those are nonisomorphic. It did 1,291 actual isomorphism checks on pairs of interpretations, involving almost

3 million permutations. Generating the OLs took about 1.4 seconds (this time was determined in a separate run, because the script `get_interps` removed that statistic), and eliminating the isomorphic ones took 2.27 seconds.

Isofilter does not attempt to permute operations when checking for isomorphism. For example, a pair of dual lattices are not necessarily isomorphic.

Get_interps. This is a simple awk script that extracts interpretations from Mace4 output. The interpretations must be in *portable format*, which is specified with the Mace4 command `set(print_models_portable)`. See the preceding `isofilter` example.

Modfilter. This program uses a set of interpretations to filter a stream of clauses. It takes two command-line arguments: (1) a file of interpretations in portable format, and (2) the type of filter to apply. The filter types are listed in the following table.

`true_in_all`	Admit clauses true in all of the interpretations.
`true_in_some`	Admit clauses true in some of the interpretations.
`false_in_all`	Admit clauses false in all of the interpretations.
`false_in_some`	Admit clauses false in some of the interpretations.

The clauses to be tested are read from the standard input, which should contain nothing but the clauses (without `end_of_list`). If the clauses need operator declarations, they should be at the beginning of the file of interpretations. The interpretations should be surrounded by "`terms(interpretations).`" and "`end_of_list.`". Here is an example that uses the ortholattices up through size 6 to filter some Boolean algebra identities.

```
% modfilter interp.OL6 true_in_all < BA.in
```

Modtester. This program takes a set of interpretations and a stream of clauses. For each clause, it tells the interpretations in which the clause is true. The calling sequence and the format of the input are similar to the program modfilter, except that no filter type is given on the command line. An example follows.

```
% modtester interp.OL6 < BA.in
```

Interpfilter. This program takes a file of clauses (as command-line argument 1), a test to apply (as command-line argument 2), and a stream of interpretations (from the standard input). The accepted tests are `models` and `nonmodels`. For each interpretation I, if I is a model or nonmodel (as specified in the test) of the set of clauses, the interpretation is passed through to the standard output. Say we have a set of ortholattices (file `OL.8`) and we wish to eliminate the distributive ones. Then we can create a file containing a distributivity clause (file `distributivity`) and run the following command.

```
% interpfilter distributivity nonmodels < OL.8
```

Otter-to-mace. This is a Perl script that takes an Otter (also Mace2) input file (from the standard input) and tries to convert it to a Mace4 input file. It is far from perfect, but is works well for many simple Otter inputs. An example follows.

```
% otter-to-mace < lattice-otter.in > lattice-mace.in
```

For more information, run the command "`otter-to-mace help`".

References

[1] W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *J. Automated Reasoning*, 9(2):147–167, 1992.

[2] W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Tech. Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL, May 1994.

[3] W. McCune. Otter 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994. Also see `http://www.mcs.anl.gov/AR/otter/`.

[4] W. McCune. MACE 2.0 Reference Manual and Guide. Tech. Memo ANL/MCS-TM-249, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, June 2001.

[5] W. McCune. LADR: Library of Automated Deduction Routines. `http://www.mcs.anl.gov/AR/ladr/`, 2003.

[6] W. McCune. Mace4: Models And Counter-Examples. `http://www.mcs.anl.gov/AR/mace4/`, 2003.

[7] J. Zhang. Constructing finite algebras with FALCON. *J. Automated Reasoning*, 17(1):1–22, 1996.

[8] J. Zhang and H. Zhang. SEM: A system for enumerating models. In *Proc. IJCAI-95*, volume 1, pages 298–303. Morgan Kaufmann, 1995.