FINDER

Finite Domain Enumerator

VERSION 3.0

NOTES AND GUIDE

John Slaney

Centre for Information Science Research
Australian National University

Version of July 20, 1995

Introduction

The program FINDER (Finite Domain Enumerator) takes as input a first order theory, expressed as a set of clauses, and gives as output the models of that theory with domains of given finite cardinality (up to a maximum size of 31). It can be used for various kinds of problem-solving, or to generate counter-examples refuting conjectures, or in combination with other reasoning systems. For example, it may help to make proof searches more efficient by providing semantic information to a conventional deduction system. For an account of what is meant by *model* in this context, see §2.1.2. For a description of some FINDER applications, see §4. For a full definition of the command language, see §2.2 and the sections following.

FINDER performs an exhaustive search for interpretations of the given language, using the given clauses as constraints to direct its backtracking. One of the main differences between FINDER and a logic programming system (which can also be seen as searching for models of sets of clauses) is that for FINDER there is no order of evaluation of the clauses[1] and hence no "flow of control" such as underlies the operational semantics of conventional systems like Prolog. A model is generated and tried against all the clauses together; if all clauses are true, the model is accepted and printed out, while if one of the clauses is false the model is adjusted to deal with the detected badness, resulting in a further candidate model. This process goes on until enough models have been found or until the search space, defined as the set of possible interpretations of the predicate and function symbols on the chosen domain, is exhausted. The other principal difference between FINDER's kind of model-generation and that associated with more deduction-oriented techniques is that for FINDER the domain of individuals is given antecedently and is not constructed from the terms of the language. The objects in, say, a three-element domain are simply the first object, the second object, and the third object. They need not have names.

The code actually used by FINDER to control the search is exactly the same as that used by the earlier program MaGIC (see [17] for details). In a

[1]This is not strictly so: of course things happen in *some* order. For description of some subtle effects obtainable by setting the evaluation order see §2 below. For the present, the right conceptual picture of the clauses is as an unordered set.

straightforward sense, the problems treated by MaGIC may be seen as special cases of those treated by FINDER, and certainly the similarities between the inner workings of the two programs run deep. However, whereas MaGIC is the product of many years of development, and is a tool for logical research aimed at a rather specific application, FINDER is rather new—two years old—and much more general-purpose. Hence substantial changes to FINDER may be expected over the releases of Version 3, though there will be an effort to retain compatibility with the present program.

If you are a user of FINDER 2 you will find that FINDER 3 is in most respects very similar. FINDER 2 input code should be usable for FINDER 3 as it stands, though there are some differences of detail which may make a rehash advisable. Some of the more obscure of FINDER 2's settings and the like are no longer supported, and generally speaking it is now a simpler matter to get acceptable performance from FINDER. Most of the differences are due to the substitution of a completely new and more efficient search algorithm. This makes the 'change-order' relatively unimportant and has the side effect that the order in which multiple models emerge is harder to predict.

In the (unlikely) event that you are familiar with FINDER 1, you should note that the differences between that and the later versions are quite large. There is no sort of compatibility between them. The most obvious differences are the introduction of many sorted logic in place of the former crude single sort and the much greater friendliness of the language used for input. Many of the settings, output formats and the like are also new, as is the possibility of piping I/O between FINDERs. Version 1 is now obsolete and should be replaced in all applications by Version 3. Conversion of any dusty decks of existing input files to the new format is not a big job.

FINDER is available by anonymous ftp from `arp.anu.edu.au` where its sources will be found in the directory `ARP/FINDER` as `finder-3.0.tar.Z` or on tape by ordinary mail from the author. See §1.2 below for installation details.

It is suggested that if you install or use FINDER you let us know that you have it, so that we can distribute information about updates, patches, etc. to as many sites as possible. Our address is:

Centre for Information Science Research
Australian National University
Canberra, A.C.T. 0200
Australia

John.Slaney@anu.edu.au
or
Zdzislaw.Meglicki@anu.edu.au

Copyright of Software

The following copyright notice and disclaimer appears in each of the source files for the program FINDER.

FINDER 3.0

(C) 1993 Australian National University,

All rights reserved.

The information in this software is subject to change without notice and should not be construed as a commitment by the Australian National University. The Australian National University makes no representations about the suitability of this software for any purpose. It is supplied “as is” without express or implied warranty. If the software is modified in a manner creating derivative copyright rights, appropriate legends may be placed on the derivative work in addition to that set forth above.

Permission to use, copy, modify and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that both the above copyright notice and this permission notice appear in all copies and supporting documentation, and that the name of the Australian National University not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Contents

1 This is FINDER **1**

1.1 What FINDER is for . . . 1

1.2 Installation . . . 5

2 Reference Guide **7**

2.1 First Order Language and Models . . . 7

2.1.1 Language . . . 7

2.1.2 Models . . . 9

2.2 FINDER input . . . 10

2.2.1 Sorts . . . 11

2.2.2 Functions . . . 13

2.2.3 Clauses . . . 20

2.2.4 Settings . . . 22

2.3 Non-standard input . . . 28

2.3.1 Piping . . . 28

2.3.2 Command line . . . 31

2.3.3 FINDER as TESTER . . . 32

3 How it Works **37**

4 Sample Applications **47**

4.1 Problem solving . . . 47

4.1.1 Jobs . . . 48

4.1.2 Squaring Up . . . 51

4.1.3 Queens . . . 55

4.1.4 Heap Arithmetic 57
4.1.5 Ordered semigroups 60
4.2 Quasigroup problems 62
4.3 FINDER in deduction 66

Chapter 1

This is FINDER

1.1 What FINDER is for

The following passage occurs in a recent document describing current research directions within the field of automated reasoning.

> The notion of reasoning comprises different kinds of inferences which are frequently used in everyday life. Deduction... is the kind of inference which has been intensively investigated for many different logics. But there are other types of reasoning which have not yet got the attention they deserve... [These include] model generation, abduction and induction. One application of model generation is the computation of counter examples for a given conjecture.
>
> J. Cunningham
> [6] §3, p. 17

Now it is indeed the case that modelling theories is a form of reasoning. Moreover, besides demonstrating that a theory has models—showing that it could be true—we might well be interested for sundry purposes in computing the content of its models—showing *how* it could be true. Consider, for example, the problem of deriving a theorem of some theory from its axioms. This is often done by breaking the goal into simpler subgoals. These subgoals, unfortunately, are often not theorems, so much time is wasted in trying to prove them. A model of the axioms in which a subgoal is false shows cheaply that no proof of it exists, so being able to find such models is of great value in theorem proving.[1] Again, consider the problem of planning a process around a set of constraints. We need models of the constraints, and are interested in

[1] For useful reflections on this use of model generating programs, see [1].

the details of specific plans more than in the existence of some plan or other. It seems that the applications of model-generating programs have only begun to be explored.

To approach the view of models as the goals of pieces of reasoning, consider a couple of examples of theories which have finite models. Let e, f and g be function symbols, e nullary, f unary and g binary. Consider the following set of sentences in a first order language with function symbols and an identity predicate.

$$\begin{array}{l} \forall x\,(gxfx = e) \\ \forall x\,(gxe = x) \\ \forall x\,(gex = x) \\ \forall x \forall y \forall z\,(ggxyz = gxgyz) \end{array}$$

It is of no interest to anybody that this set has a model. Since it is a set of equations it has a trivial model in a universe of just one object. But of course its models, better known as *groups*, are the subject of intensive study and constitute one of the most important topics in mathematics, so the *content* of its models is of great interest while the *existence* of models is not. We might well, therefore, set out to generate and examine the small models of such a theory.[2] The program FINDER is intended for just such purposes.

Group theory is in many ways special, so that techniques for modelling arbitrary sets of sentences would be ludicrously cumbersome as methods of generating groups, but there are many other kinds of algebra which can be approached quite usefully by general methods. Consider totally ordered semigroups, for example.[3] There are many more of these of a given finite size than there are groups, and they are much less orderly. Assuming that the elements of our domain are the first few natural numbers (which are as good as anything for the purpose and come conveniently ordered) the postulates are these:

$$\begin{array}{l} \forall x \forall y \forall z\,(x \leq y \;\Rightarrow\; fxz \leq fyz) \\ \forall x \forall y \forall z\,(x \leq y \;\Rightarrow\; fzx \leq fzy) \\ \forall x \forall y \forall z\,(ffxyz = fxfyz) \end{array}$$

Here f is the semigroup operation, of course, and we are assuming that the relation $\leq$ is primitive. These postulates have, for instance, 42640 models in a domain of six elements, and the computer generation of these for purposes of study as well as for purposes of refuting nontheorems of ordered semigroup theory is entirely feasible and worthwhile.

[2] Lest it be thought that generating finite groups would also be of little interest, since they are so well known, note that the generation and characterisation of the groups of order 128 was a particularly recalcitrant problem solved only recently at ANU (not by the Automated Reasoning Project but by our colleagues in the Mathematics Department).

[3] This problem is used as an example throughout the present notes, not because I think it is a deep or important problem but because it is clear, simple to describe and fairly tough to solve.

A program for generating finite models of sets of clauses has many applications besides algebra. For example, one justly famous problem is that of designing a timetable assigning activities to persons and locations and times in such a way that every person gets to engage in his or her preferred set of activities, nobody has to be in two places at once, no two activities are in the same place at the same time and so forth. These constraints can be expressed as simple first-order sentences which must all be true of any model proposed, and in general a suitable method of solution is to *generate* candidate timetables and *test* them for satisfaction of all the constraining conditions, continuing until enough satisfactory timetables have been found. Crude generate-and-test algorithms rapidly run into an explosion in the size of the search space as the number of variables requiring values increases. More sophisticated versions, however, such as that used in FINDER, can at least mitigate the effects of this explosion so that worthwhile instances of such problems can be solved in a reasonable time.

For a final, more amusing example of a problem suitable for FINDER, here is a logical puzzle known as the Philosophical Railway Problem.

> It happened, in the days when trains used to call at the tiny village of Much Tittering in the Woods, that the 1215 once pulled up there and stood for the best part of an hour. Nobody now remembers why. At any rate, the driver, the porter, the ticket inspector, the stationmaster and the guard spent the time in such merry conversation as is customary among employees of railway companies. Their names, in alphabetical order, were James, Kant, Locke, Mill and Nietzsche.
>
> For reasons lost in the mists of railway history, they agreed to make two statements each, one true and the other false. They said:
>
> MILL: Nietzsche is the stationmaster.
> James is either the guard or the porter.
>
> LOCKE: Neither Kant nor Nietzsche is the ticket inspector.
> Mill is not the stationmaster.
>
> KANT: Mill's second statement was false.
> Locke's first statement was true.
>
> NIETZSCHE: Either James is the porter or I am.
> Neither Locke nor Mill is the guard.
>
> JAMES: I am not the ticket inspector.
> Nietzsche's second statement was false.
>
> What was the driver's name?

Now if we are thinking clearly we shall see that this reduces to a problem of permuting the five occupations until we find one permutation that makes exactly one of each pair of sentences true. There are, in fact, several ways in which the problem can be put to FINDER, some of which are more efficient than others. Most straightforward is to regard the philosophers and their occupations as different sorts, each listed by the given names, and to generate a bijection between them satisfying the constraints. In this form, it occupies FINDER for about 50 milliseconds on a SPARC-2, most of that time being taken up with reading in the problem, pre-processing it and printing the results.

So the range of problems usefully addressed by a program which searches for finite models of sets of first order sentences is really quite large. Some of the problems which FINDER can solve would be better addressed by a program which looks for models in some other way such as letting them occur as a by-product of a search for a proof that there are none. There is a body of recent work on uses of resolution-style provers to generate models, and some results with constraint logic programming systems are very encouraging indeed. Such overtly deductive methods would seem particularly applicable to cases in which the models are very rare or even unique, and in which the clauses specifying them have a high degree of recursiveness. There are other cases, however, in which a specialised and sophisticated searching method such as that of FINDER is preferable to a more deductive technique. For example, staples of constraint satisfaction such as the Queens Problem of enumerating the ways of placing n queens on an $n \times n$ chessboard so that no queen attacks any other tend to be easier for FINDER than for typical applications of logic programming methods. Ultimately, then, FINDER is recommended not as the ultimate solution to all known problems but as a tool among others to be applied sensibly as the occasion warrants.

1.2 Installation

This section is addressed to the system administrator of a Unix[4] installation. It is duplicated in the README file supplied with FINDER. If you have any problems installing the program contact the author.

In order to install FINDER 3.0 you should have a directory containing the following files.

```
FINDER.c
FINDER.h
Fdef.h
Fglob.h
Fmain.c
Fprotos.h
Ftypes.h
Makefile
README
finder.man
initial.c
input.c
newtest.c
output.c
parser.c
prepare.c
pretest.c
set_space.c
test.c
newtest.c
vntr.c
vntr.h
```

If you get the program by anonymous ftp from our site, after uncompression you will have these files in a directory, and also a subdirectory `./doc` containing the LaTeX source for the full documentation, and another subdirectory `./samples` containing some sample input files. Before compilation and installation edit the Makefile in the main directory of FINDER. On most systems only the top few lines of the Makefile will require editing. `$(BIN)` defines where FINDER's binaries will go. The manual will be installed in `$(MAN)/man$(MAN_EXT)` as `finder.$(MAN_EXT)`.

On our system binary files and manual pages are writable to the group. This is reflected in variables `MANDMDE`, `MANMODE` and `BINMODE`—modify these if you don't want the group to be allowed to write on the binaries and libraries.

[4]'Unix' is a trade mark of AT&T Bell Laboratories.

To read the command line FINDER uses AT&T's function `getopt`. Under certain operating systems this means you will have to use special libraries during linking. For example, under Dynix on a Sequent you will need the −lseq library. Alternatively, get GNU's `getopt` and link it with FINDER if you don't have it on your system already.

FINDER will not compile with Kernighan and Ritchie `cc` so you should probably leave `CC` defined as `gcc`. Alternatively, you may use any other ANSI-C. We have not tried this ourselves, however, so it may not work.

Once you are happy with the Makefile, type

```
make
```

If there were no problems with compilation and linking, type

```
make install
```

to install the binaries and manuals. Then to clean up the source directory type

```
make clean
```

Happy FINDing!

Chapter 2

Reference Guide

2.1 First Order Language and Models

2.1.1 Language

The formal language accepted by FINDER is that of a many-sorted first order logic. Sorts are not indicated by any markers in the surface syntax, such as subscripts or different styles of variable, but must be used consistently for formulas to be well formed. Formulas must be Skolemized and reduced to clause form, but the clauses need not be Horn. Function symbols must be pre-declared before they are used in clauses, but variables need not be. Any legal identifier (see below for a definition) may be used as a function symbol or as a variable. Canonical names are available for objects of enumerated sorts: again see §2.2.1 below for details.

The formation rules are as follows.

1. A variable is a formula.
2. A canonical name is a formula.
3. Where $X_1 \ldots X_n$ are formulas of sorts $s_1 \ldots s_n$ respectively, and where f is a function symbol of type $s_1, \ldots, s_n \rightarrow t$ the compound expression $f(X_1 \ldots X_n)$ is a formula of sort t.
4. Where X and Y are formulas of the same sort, each of $X = Y$, $X < Y$ and $X > Y$ is a formula of sort `bool`.
5. Where X is a formula, E! (X) is a formula of sort `bool`.
6. Where X is a formula, LEN (X) is a formula of sort `int`.

7. Where X is a formula of sort s, and n is a natural numeral, $X + n$ is a formula of sort s.

8. A clause is an ordered pair of sets of formulas.

A variable may be of any sort, but must be of the same sort throughout a clause. The pre-defined sort `bool` consists of the two objects `false` and `true`. The pre-defined sort `int` consists of the first few natural numbers (normally $0 \ldots 31$) which have the expected canonical names.

A few conventions govern the writing of clauses to FINDER. There is one fairly severe limit in that no function symbol may be of arity greater than 2. Clauses must be finite in order to be written, of course, though there is in principle no limit on the number of formulas a clause may contain.

Dyadic function symbols may be placed in infix position or in prefix position at the user's whim. If prefix notation is used, parentheses are needed around the arguments and a comma between them. Thus FINDER will parse

```
Tom loves Mary.
```

correctly, though if you feel that

```
loves(Tom,Mary).
```

looks "more logical" or something, then you may use it instead. You may even mix prefix and infix notation in the same clause. Monadic function symbols are always prefixed to their arguments, which are always parenthesized. There are no conventions about the difference between upper case and lower case letters. This feature contrasts with languages such as Prolog, where upper case initial letters (perversely) indicate variables. FINDER is case-sensitive everywhere, however.

As usual where infix notation is used, there are conventions allowing parentheses to be dropped in some circumstances. Dyadic function symbols have "scopes" (see §2.2.2) and the general convention is that where f has greater scope than g

`X f Y g Z` is parsed as `X f (Y g Z)`

`X g Y f Z` is parsed as `(X g Y) f Z`.

Among function symbols of the same scope, association to the left is assumed. Excess parentheses are ignored, but they must match in pairs.

A clause $\langle A, C \rangle$ is written by listing the members of set A, separated by commas, then an arrow `->` and then the members of C also separated by commas. A period terminates the clause. A is the set of antecedents (negative literals) and C the set of consequents (positive literals). The order of formulas within each list is irrelevant. Either the antecedent list or the consequent list (but not both) may be null. Where the antecedent list is null and there is only one consequent formula, the arrow may be omitted.

2.1.2 Models

We now consider a first order language with a set S of sorts, a set F of function symbols and a set V of variables. A model of this language consists of two functions. There is a function $\mathcal{D}$ assigning to each sort a well-ordered nonempty set called its domain, and a function $\mathcal{I}$ assigning to each function symbol an appropriate function. The condition to be met is the expected one, that if

$$f:\ s_1 \times \ldots \times s_n \longrightarrow t$$

then

$$\mathcal{I}(f):\ \mathcal{D}(s_1) \times \ldots \times \mathcal{D}(s_n) \longrightarrow \mathcal{D}(t)$$

As a refinement, where function symbols are marked as "partial" the functions assigned to them are allowed to be partial as well.

Next, a valuation $\mathcal{V}$ is a function assigning to each variable of sort s an element of $\mathcal{D}(s)$. In terms of this we may define denotation δ relative to a given $\mathcal{D}$, $\mathcal{I}$ and $\mathcal{V}$:

1. For $v \in V$, $\delta(v) = \mathcal{V}(v)$
2. $\delta(f\, t_1 \ldots t_n) = \mathcal{I}(f)\,(\delta(t_1) \ldots \delta(t_n))$

That is on the assumption that each $\delta(t_i)$ exists. If one of the functions is partial, so that $\delta(t_i)$ does not exist, then $\delta(f\, t_1 \ldots t_n)$ does not exist either. The formula's denotation is then said to be undefined on valuation $\mathcal{V}$. Exceptions are made for the constant symbols for identity, order and existence:

1. $\delta(t = u) =$ `true` if $\delta(t) = \delta(u)$ or if both $\delta(t)$ and $\delta(u)$ are undefined. Otherwise $\delta(t = u) =$ `false`
2. $\delta(t < u) =$ `true` if $\delta(t) < \delta(u)$ or if $\delta(t)$ is defined and $\delta(u)$ is not. Otherwise $\delta(t < u) =$ `false`
3. $\delta(t > u) =$ `true` if $\delta(t) > \delta(u)$ or if $\delta(u)$ is defined and $\delta(t)$ is not. Otherwise $\delta(t > u) =$ `false`
4. $\delta(\mathrm{E!}\,(t)) =$ `true` if $\delta(t)$ is defined. Otherwise $\delta(\mathrm{E!}(t)) =$ `false`

In addition, $\delta(\mathrm{LEN}\,(t))$ is the length of the canonical name of $\delta(t)$.

Valuation $\mathcal{V}$ falsifies a formula X of sort s iff $\delta(X)$ is $0_{\mathcal{D}(s)}$, the zero element in the well ordering of the appropriate sort. Otherwise it verifies X. $\mathcal{V}$ verifies a clause $\langle A, C\rangle$ iff either $\mathcal{V}$ either falsifies some member of A or verifies some member of C. $\langle A, C\rangle$ is true in model $\langle$D,I$\rangle$ iff it is verified by every valuation associated with the model.

So a clause is a disjunction. The antecedents are treated as negated and the consequents as un-negated. Note that any nonzero value, even being undefined, counts as truth for the purposes of verification. Note also that the formulas in a clause do not have to be of sort `bool` although they usually will be. As usual, all variables are regarded as bound by universal quantifiers standing outside the clause. In this definition of models, variables have sorts. This is really so in the syntax as written, except that the sorts are deduced by FINDER from the context rather than being made explicit.

In practice, of course, all the domains are finite. They are *very* finite in fact, usually containing only a handful of individuals and never more than the maximum of 31.

2.2 FINDER input

A FINDER input file consists of one or more *sections*. There are four types of section, each consisting of a key word followed by a specification list. The key words are

```
sort

function

clause

setting
```

There is no limit on the number of sections of each type and no restriction on the order in which they may occur, except that any function symbol used in a clause must be declared before it is used, and any sort used in a function declaration must likewise have been declared before it is used.

End of input is indicated by placing the word

```
end
```

after the last specification section in the file. Any text following this is regarded as a comment.

A specification list consists either of a single specification (e.g. a function declaration or a clause) or of a sequence of them enclosed in braces. There are no conventions about placing specifications one to a line or about indenting, etc. Any white space is regarded as a separator. In general, punctuation marks such as commas, periods and semicolons are not required to separate

items in a specification list, though a period in a list will be treated as the null specification. There are some obligatory uses of punctuation, however, and some optional ones, for which see the detailed accounts below.

Comments may occur anywhere in an input file. A comment begins with the reserved character `%` and is terminated by a line feed. Any text in between is totally ignored by FINDER. Intelligent use of comments, as well as sticking to some reasonable conventions about indentation and the like, greatly aids readability.

FINDER reads its input from the standard input stream and writes its output to the standard output stream. There are special cases in which it can be directed to take input from a specified file as well: see sections §2.3.1 and §4.3 below for details.

A legal identifier for FINDER purposes is any string of characters not containing white space, with the exception of certain reserved words and symbols as follows. Any string beginning with a numerical digit is taken to be a numeral and is not available for any other purpose. The characters

`+`   `=`   `<`   `>`

are reserved for special purposes and may not occur inside identifiers other than themselves. Punctuation symbols

.   ,   :

may not occur in identifiers, and nor may (round) parentheses or {curly} braces. The "percent" symbol introduces a comment and may not be used for any other purpose. Finally, overloading is not allowed. Although FINDER is case sensitive, so for example you can use `THING` as a sort name, `Thing` as a function symbol and `thing` as a variable all in the same file, you cannot use a word both as a sort name and as a function symbol in the same input. Nor can you have two different functions with the same name.

If a passage of input contains inconsistent specifications (for example a line setting the output style to "pretty" and one setting it to "ugly") then the specification occurring later over-rides the earlier one. This becomes especially important when command line input is used (see §2.3.2 below).

2.2.1 Sorts

Any legal identifier may be stipulated to be a sort name. It then picks out a domain of individuals. There are two pre-defined sorts, `bool` and `int`. It is not a good practice to use these for purposes other than representing truth values and natural numbers respectively.

At any given time during FINDER's search for models, each sort has a particular cardinality. The minimum cardinality is 1 and the maximum is a figure set when FINDER is compiled. By default it is 31 (the maximum possible, for no especially good reason). The range within which the cardinality of a sort must fall may optionally be restricted by inserting specifications immediately after its declaration. A cardinality specification consists of the word `cardinality` followed by a numerical comparator (`=`, `<` or `>`) followed by an integer. Thus for example

```
sort {
    dwarf
        cardinality = 7

    princess
        cardinality < 4

    servant {
        cardinality > 1
        cardinality < 6
    }

    wolf {}
}
```

sets up a story in which there are seven dwarves (of course), at most three princesses, a plurality of servants, but not more than 5, and we don't care how many wolves. Every sort name is followed by a specification list: in the case of the wolves it is the empty list, but it must still be given.

The order in which the sorts are presented is significant. FINDER searches spaces of possible models determined by setting the cardinalities of the sorts. Each different selection of cardinalities from within the stipulated ranges determines a separate search space. The order in which the search spaces are considered depends on the order of sorts. The sorts presented first are most significant (change their cardinalities least often) and those presented last are least significant. Lower cardinalities (smaller domains) precede higher ones. Thus one search space comes before another if they have the same cardinalities for the first $n-1$ sorts and the first space has a lower cardinality for the nth sort than does the second.

The pre-defined sorts come first of all (`bool` before `int`) and have default cardinalities of 2 and 31 respectively.

An alternative to stipulating the cardinality of a sort is to list the objects of that sort, giving each a canonical name. The enumeration replaces the cardinality specification immediately following the sort declaration, and consists

of the key word `enum` followed by the list of names separated by commas and terminated by a period. For example

```
sort dwarf
   enum: Happy, Sleepy, Dopey, Doc, Sneezy, Grumpy, Bashful.
```

The colon after the `enum` is optional. The canonical names supplied in an enumeration serve as symbolic constants for the purposes of syntax. It is generally more efficient to give names by enumeration than to declare them as zero-adic function symbols, although their syntactic status in clauses is the same.

According to FINDER, to enumerate objects is to put them in one-one correspondence with the first few natural numbers, starting with zero. Some numerical properties are carried over to enumerated sorts. The canonical total order is well defined on every domain, and in the case of an enumerated sort it is given by the enumeration order. The initial item (`Happy` in the above example) is the zero of sort dwarf, and in accordance with the C-like underlying semantics is the only dwarf not regarded as "true". As is made clear in §2.2.2 the operation of addition is defined on every sort, so for example `Sleepy + 3` is, by the definition encoded in the enumeration, `Sneezy`.

The cardinality of an enumerated sort may not be changed by subsequent cardinality specifications, since the enumeration itself fixes it. An exception is made for the sort `int` which is enumerated, with the obvious canonical names, but which may be truncated as desired. The other pre-defined sort, `bool`, is also enumerated:

```
sort bool
   enum: false, true.
```

Every sort has a dummy object which is taken to be the value of a partial function in cases where it has no value of the ordinary sort. This object cannot be the value of a variable. That is, it does not exist for normal purposes. It has no name. The dummy object is a kind of infinity, being greater than any thing in the domain. The arithmetical comparators apply consistently to it on that basis. Any other function with the dummy as an argument gives the dummy as value. In most FINDER applications, only total functions are required, and so these "junk" objects need not be taken into account. There are cases, however, in which partial functions are useful, so the facility is provided.

2.2.2 Functions

Any legal identifier not already in use may be declared as a function symbol. Functions may be nulladic (in which case the symbols are proper names),

monadic or dyadic. Function symbols of higher adicity are not allowed in FINDER at present, though they should be added in future releases. A function specification consists of the function name, followed by its type definition, followed by a specification list giving some of its features. The order in which functions are defined is important, as it can affect the efficiency of FINDER's search. FINDER backtracks its way through the search space changing the last-declared function most rapidly and the first-declared least rapidly.

As explained in §2.1.2 each function maps a Cartesian product of sorts (its argument sorts or domain) into one sort (its value sort or co-domain). The type definition in a function symbol declaration consists of the list of argument sorts, ordered left to right and separated by commas, then an arrow `->` followed by the value sort. In the special case of a zero-adic sort, where the list of argument sorts is null, the arrow may optionally be omitted. In that case, the value sort must be followed by a period, to tell FINDER that the type definition is finished. That is,

```
function Snow-White
   -> princess
```

is entirely equivalent to

```
function Snow-White
   princess.
```

An alternative notation, requiring neither arrow nor period, is

```
constant Snow-White
   princess
```

A predicate or relation symbol normally has `bool` as its value sort, though nothing prevents us from setting up a many-valued system such as the following one for the three-valued logic RM3.

```
sort  RM3-value
   enum: False, True, TRUE.

function  implies
   RM3-value, RM3-value  ->  RM3-value
```

Notice, however, that in evaluating clauses FINDER will treat all values except the initial "zero" one as designated. If we wish to have several undesignated values we shall have to add a function `designated` from values to `bool` after all.

Several function symbols are pre-defined. Every sort `S` comes equipped with the following.

```
function {
    = : S,S -> bool {}

    < : S,S -> bool {}

    > : S,S -> bool {}

    E! : S -> bool {}

    + : S,int -> S { partial }

    LEN : S -> int {}
}
```

The first three of these are the comparators which, as noted in §2.2.1 above, give truth values even with dummy arguments. The fourth is the "existence" predicate for sort S. It gives the value `true` when its argument exists and `false` when its argument is the dummy. The addition function allows any object of sort S to be incremented by an integer. The result may be too big to exist, so this is a partial function. Finally, `LEN` returns the number of characters in its argument's canonical name. FINDER calculates this anyway, for tabulating its output, so it costs little to make it available in the syntax.

The specification list of a function declaration may be quite complex. The various options are treated in the following subsections.

Total and partial

Functions may be total, in which case they have a value of their value sort for each tuple of arguments, or partial, in which case certain arguments may result in no value. FINDER uses the convention, first suggested apparently by Frege, that all terms which fail to refer in the normal way be taken to refer to a "junk" object. As noted in §2.2.1, each sort has its own dummy object which functions as a kind of infinity for that sort. Where partial functions are used, the existence predicate `E!` will be found valuable. For an example, see the formulation of the Queens Problem in §4.1.3.

The single word `total` (or `partial`) specifies a function as total (or partial). Apart from the pre-defined addition function, all functions are total by default.

Injective and surjective

Many problems turn on the fact that no two arguments for a function yield the same value. In this case the function is said to be an injection. This condition

is so common and so special that FINDER has a hard-coded efficient way of dealing with it. The complementary property to injectiveness is surjectiveness: a surjective function is one such that every object of the value sort is the value of the function for some argument or arguments. FINDER also has reasonably efficient ways of dealing with surjections. An injective function, then, is one-one while a surjective function maps the argument sort(s) onto the value sort. A function which is both an injection and a surjection is a bijection. It correlates the argument domain with the value domain exactly.

The words `injective`, `surjective` and `bijective` are available as items in function specification lists.

Some interesting dyadic functions are not injective in the above sense, but are injective (or surjective, or bijective) in the left argument place, the right argument place or both. We say that dyadic function f is left injective, or column injective, iff for all arguments x, y and z if $f(x,z) = f(y,z)$ then $x = y$. Right (or row) injectiveness, left and right surjectiveness and left and right bijectiveness are similar.[1] A dyadic operation on a domain which is both left bijective and right bijective is a quasigroup operation. Its Cayley matrix has the property that every row and every column is a permutation of the elements. Some hard problems involving quasigroups are considered in §4.2.

The expressions `left-injective`, `right-injective`, `column-injective`, `row-injective`, `left-surjective`, `right-surjective`, `column-surjective`, `row-surjective`, `left-bijective`, `right-bijective`, `column-bijective` and `row-bijective` are available as items in the function specification lists of dyadic function symbols only.

Commutative

The word `commutative` is available as an item in the function specification lists of dyadic function symbols only. It stipulates the property that $f(x,y) = f(y,x)$ for all arguments x and y. FINDER has an efficient way of forcing commutativity, so this specification should always be used when it applies, in preference to a clause laying down the commutativity axiom.

Cut

The word `cut` may be placed in a function specification list to control backtracking rather as in Prolog and similar systems. If a function symbol is marked

[1] An operation with one of these injectiveness properties is usually said to be *cancellative* in the appropriate sense. The only defence for inventing new terms here is that they help to keep the vocabulary uniform.

with a cut, FINDER will not generate two models whose most significant difference is in the values for the marked function. Thus for example, to generate failures of commutativity in semigroups we may simply declare two Skolem constants and stipulate that the semigroup operation does not commute on them:

```
function  *: int,int -> int {}

constant {
    a: int {}
    b: int {}
}

clause {
    x * y * z = x * (y * z).
    a * b = b * a -> false.
}
```

The following, by contrast, generates non-commutative semigroups

```
function   *: int,int -> int {}

constant {
    a: int { cut }
    b: int { cut }
}

clause {
    x * y * z = x * (y * z).
    a * b = b * a -> false.
}
```

To turn off a cut, the specification `no-cut` is provided. Note that a cut is incompatible with the no-priority specification, so an implicit `priority` immediately follows the word `cut` in the input, and conversely `no-priority` is implicitly followed by `no-cut`.

Print mode

The print mode of a function is specified by the word `print` followed by one of the three modes `none`, `brief` or `full`. A setting of `none` causes printing of the function to be suppressed whatever global print mode for models has been specified as a setting (see §2.2.4 below). A setting of `brief` causes the function

to be printed out only when its value, or that of a more significant function, changes. A setting of `full`, naturally, causes the value table for the function to be printed out whenever a model is found. The brief and full settings have no effect if over-ridden by low global print modes for models.

The default print mode for the pre-defined functions is `none`. For other functions it is `full`.

Scope

Each function symbol has a scope, which is a number in the range 0...10. Symbols of larger scope bind less tightly than those of smaller scope for the purposes of the parser. Scope conventions allow most parentheses to be omitted where dyadic function symbols are written in infix notation. The default scope is 0 for monadic symbols, 1 for ordinary dyadic ones and 10 for the pre-defined arithmetical comparators.

Scope is specified with the word `scope` followed by an integer.

Change-order

When FINDER is backtracking its way through the search space, it proceeds by choosing a function and a possible argument for it, setting the value of the function for that argument (or argument tuple) to each possibility in turn and in each case searching the subspace which remains while that value is held fixed. How does it choose which function and argument to fix next? This is a little complicated, so pay attention!

For brevity, let us refer to a function-argument pair as a *cell*. If there is a cell with only one possible value, then this value is inserted next no matter what else happens. If not, the program has to choose a cell for splitting. In general, it chooses a cell pertaining to a more significant function, as determined by the order of declaration, over a less significant one. This preference may be suppressed by means of the `no-priority` specification which puts successive functions on a par for change order purposes. Within each function (or no-priority block of functions) it next prefers a cell with the smallest number of possible values. This is for reasons of efficiency, to minimise furcation of the search tree. Among cells of equal priority and with equal numbers of values it chooses the first in the 'natural' change order for the function (or first of the functions in the block). The natural change order for any function may be set by means of the `change-order` option, though in practice it is usually sufficient to allow FINDER to set the order by default.

For monadic functions there is only one available order, taking the arguments in order of size. The change order is either `+` in which case lower

arguments are less significant, or `-` in which case lower arguments are more significant than higher ones.

For dyadic functions things are more complicated. Consider what the matrix for a sample dyadic function `f` might look like as printed by FINDER:

```
 f | A B C D E F
---+------------
 A | A B C D E F
 B | B F E C D A
 C | C E F B A D
 D | D C B A F E
 E | E D A F B C
 F | F A D E C B
```

There are 36 cells in this diagram which must be filled with values. FINDER puts them in order of significance by means of the following algorithm. At any stage, we let `R` be the set of cells which have no position as yet in the significance order. The cells in `R` form a rectangle of the printed version.

```
p := 0
R := all of the cells
repeat
   choose one edge E of the rectangle R
   for each cell C in E do
      set significance of C to p
      p := p+1
   end for
   R := R-E
until R is empty
```

The two choice points here are to determine (a) which edge of the remaining rectangle to treat next, and (b) whether to take its cells in ascending or in descending order. FINDER makes these choices by looking at the change order string associated with the function. This consists of a sequence of edge specifications: `T` or `t` for the top edge, `B` or `b` for the bottom, `L` or `l` for the left and `R` or `r` for the right. Upper case means take the cells of that edge in increasing order, while lower case means take them in decreasing order. When FINDER gets to the end of the string of letters it simply cycles back to the beginning. As in the monadic case, we stipulate `+` to assign significance values in increasing order, or `-` to assign them in decreasing order.

The default for monadic functions is `change-order +` and the default for dyadic functions is `change-order +T`. This results in the following natural change order for the example of function `f` above.

```
   | A  B  C  D  E  F
---+------------------
 A |  0  1  2  3  4  5
 B |  6  7  8  9 10 11
 C | 12 13 14 15 16 17
 D | 18 19 20 21 22 23
 E | 24 25 26 27 28 29
 F | 30 31 32 33 34 35
```

The lower-right corner is the most significant cell, and the change order works its way back to the upper-left in row-major fashion. This is adequate for most purposes. An alternative, used in §4.2 for instance, is `change-order -rb`. In this order, rows and columns alternate in herringbone fashion:

```
   | A  B  C  D  E  F
---+------------------
 A |  0  2  6 12 20 30
 B |  1  3  7 13 21 31
 C |  4  5  8 14 22 32
 D |  9 10 11 15 23 33
 E | 16 17 18 19 24 34
 F | 25 26 27 28 29 35
```

Another variant, causing FINDER to fix the middle values first and spiral outwards, would be `change-order +TRbl`:

```
   | A  B  C  D  E  F
---+------------------
 A |  0  1  2  3  4  5
 B | 19 20 21 22 23  6
 C | 18 31 32 33 24  7
 D | 17 30 35 34 25  8
 E | 16 29 28 27 26  9
 F | 15 14 13 12 11 10
```

Remember that during the search, high numbered cells get their values inserted first.

2.2.3 Clauses

A FINDER input file will normally contain at least one `clause` section. Such a section consists of the key word `clause`, followed optionally by a parenthesized

integer, followed by a specification list of clauses. For an account of the meaning of clauses and for their syntax see §2.1.1 above. Note that every clause must end with a period.

To understand the subtleties of clause input, it is necessary to know something of how FINDER uses clauses to constrain its search for models. Every clause has associated with it a *position*, a *priority*, a *variable count* and a *weight*. These are non-negative integers. The weight is a measure of the information conveyed by the clause. It is defined as the maximum number of values for (non-constant) functions which have to be determined in order for the clause to be evaluated in a model. This is the number of distinct subformulas in the clause, excluding variables, canonical names and pre-defined function symbols. The variable count is just the number of variables in the clause. The priority is specified by the parenthesized integer after the word `clause` at the start of the section. Hence, to use differences of priority multiple clause sections are necessary. If no priority is specified, the default of zero is assumed. The clause positions simply record the order in which the clauses occur in the input. The first clause has the lowest position number.

Before generation and testing commences, FINDER re-orders the clauses. To do this it uses the two settings `pre-test` (default 3) and `test-by-...` (default `test-by-vars`). Clauses of weight pre-test or less are treated at a pre-processing stage, while constraints from heavier clauses are found by 'lazy constraint generation' during the generate-and-test phase. 'Test-by-vars' makes the variable count more significant than the weight in determining which clause to test next during generate-and-test; 'test-by-weight' makes weight more significant. Normally there is no need for the user to change these parameters from the defaults.

Clauses of low weight (up to the pre-test limit) come before clauses of high weight. Within the low weight class the order of clauses is determined by weight (low first), then variable count (low first), then priority (high first). Within the high (testable) weight class, priority is the most important factor, followed by variable count, and weight, or weight and variable count if 'test-by-weight' has been specified. Finally, accession order resolves any remaining cases.

Any candidate model tested by FINDER for satisfaction of all the clauses will already be guaranteed to satisfy any clauses of weight up to the pre-test limit. The remaining clauses are tested one by one. For most clauses, the test is a matter of crudely assigning all possible combinations of values to the variables occurring in the clause and looking up the consequent values of all the subformulas in each case. Hence clauses with many (more than about 5 or 6) variables will greatly slow down the search. Clauses with more than 7 variables are tested more intelligently by regarding the search for a falsifying

assignment of values as another constraint satisfaction problem and calling the FINDER search routine to generate it.

As soon as a clause fails the test, the combination of values responsible for the failure is recorded and steps are taken to avoid incorporating that particular combination ever again in a candidate model. Testing of the particular candidate then stops, no further clauses being examined, and a new candidate is generated. Thus clauses later in the order will not be examined unless the structure satisfies those earlier.

As is usual in reasoning systems of all kinds, logically equivalent formulations of a problem can produce markedly different behaviour. In general it pays to look for formulations which keep the weights of clauses low, and clauses containing many variables are to be avoided if at all possible. Since there is no concept of "flow of control" in FINDER, the priority and position orders are not usually of much significance, but there are cases in which they really matter. One such case is the 'Heap Arithmetic' example in §4.1.4.

2.2.4 Settings

The `settings` section allows a number of miscellaneous parameters to be specified. The most commonly used settings concern the various output options, but there are several others. Some of these are only used to control fairly obscure features of the search algorithm. They have default values which are good for nearly all purposes, and it is recommended that the defaults be used everywhere unless you are a FINDER expert. All settings, in fact, are optional.

Number of backtracks

The setting specification `backtracks` followed by a non-negative integer sets a limit on the number of times the search is allowed to backtrack before FINDER cuts its losses and stops searching. The default setting is 0, which sets no limit. Backtracks occur when model candidates are tested (whether or not they pass the test), when the look-ahead facility of the search controller detects that some cell has no possible value left, or when examination of a surjective function shows that some value is no longer possible for it. As an approximate guide, FINDER might be expected to backtrack between 5 and 10 times per million instructions (that is, for example, 50–100 backtracks per second on a 10 MIPS machine).[2] This figure is *very* rough, and deviations from it of up to an order of magnitude may occur.

[2] It performs only integer computation, so the floating point performance of your computer is irrelevant to this figure. MIPS (otherwise Meaningless Instructions Per Second) can be used as a good guide here.

Current priority

When FINDER is used as a "tester" adjunct to another program, each clause that is input is labelled as the "current clause". It is added to the theory only under certain conditions. The setting specification `current-priority` followed by a non-negative integer sets the priority of the current clause. Clauses given in advance may have any priority. Those added during the search have priority 0. The current clause by default has priority 1, though the `current-priority` setting allows this to be changed.

Noisy or quiet

The one-word setting `noisy` causes the terminal bell to sound every time a model is printed. The setting `quiet` (the default) turns off this action.

Pre-test limit

As explained in §2.2.3 above, clauses of low weight are treated by the preprocessor while those of high weight are set aside until the generate-and-test phase. The weight is the number of function symbols in the clause, not counting the built in ones such as '=' and 'E!'. The boundary between "low" and "high" for this purpose is by default 3 (almost always the optimal figure) but may be changed by means of the setting `pre-test` followed by an integer, being the maximum weight of pre-testable clauses. Note that in calculating this weight FINDER regards constants (0-adic function symbols) as if they were canonical names, not requiring table look-up, if their values are fixed. If their values can vary, of course, they are counted in the weight like any other function symbols.

Pretty or ugly output

There are two formats for output. The one-word setting specification `pretty` causes output that is easily human-readable. Models are numbered, everything is labelled and the values are neatly tabulated. This is the default format. For some purposes, especially where output is to be piped through another program, the buttons and bows of pretty output would get in the way, so an ugly format is preferable. The one-word specification `ugly` causes output of this form. It also has side effects of setting the verbosity levels to

```
verbosity {
    job: brief. models: brief. stats: none.
}
```

For the detailed syntax of ugly output, see §2.3.1.

Primary and secondary

As explained in §3, the search controller of FINDER uses a particular kind of inference rule to deduce "secondary" constraints which help to guide its search in addition to the "primary" ones which correspond to ways in which the clauses could be false. Usually this processing of secondary constraints greatly speeds up the search, but in a few cases it actually slows things down. Well, in one case, to be exact: the Queens Problem. Just for such pathological problems, the setting specification `primary` is provided. It causes processing of secondary constraints to be suppressed altogether. Normal, healthy problems do not need it. The setting `secondary` is the negation of `primary` and is on by default. It may be used, for instance in the command line, to override a `primary` setting in file input.

Refutation length

Another exotic option allows you to set a maximum refutation length. See §3 for an account of the negative constraints called "refutations" which FINDER uses to drive the search. The length or cardinality of a refutation is the number of cell-value pairs it involves. If this number is large, the refutation is of little value as it gives little information and uses up a large chunk of the database. It is also likely to be subsumed by a smaller and therefore more informative refutation later in the search. FINDER cuts its losses on processing large refutations by pretending that they are small ones and then throwing them away when this pretence is no longer safely sustainable. The maximum refutation length is the boundary between "large" and "small" refutations for this purpose. It has a default value of 6, and may be reset to any positive value by means of the setting specification `ref-length` followed by a positive integer. In practice this option is hardly ever used except when FINDER is fine-tuned for a specific problem set. In that case, experimentation is the only way to discover the optimum setting. A setting of more than 8 produces a warning message, since it is in principle possible for too large a setting to crash the program. In practice, this has never happened, but our insurance policy says you are to be warned.

Reporting reasons for breaks

Normally if FINDER breaks off a search before the search space is exhausted it prints the reason (e.g. it has found enough solutions or the time limit has expired). The setting `silent-breaks` turns off this reporting of reasons, and `report-breaks` (the default) turns it on. Where FINDER is called as a procedure from another program, for example, intrusive reports of this kind may not be desired, so this setting is provided to disable it.

Reverse test

Independent of the change order (see §2.2.2) is the order in which FINDER tries assigning values to variables in clauses when it is testing for truth in candidate models. Normally it tries high values for the variables before low ones, as refutations with higher variable values tend to involve cells which are more significant in the default change order and therefore to be slightly more efficient. The option `reverse-test` causes low values to be assigned first instead. See the Heap Arithmetic example in §4.1.4 for an illustration. The setting `obverse-test` is the default and causes high values to be assigned first.

Number of solutions

The setting specification `solutions` followed by a non-negative integer sets the maximum number of models required. The most usual settings are 1 (stop when a single model is found) and 0 (the default, no limit). It can also be useful to set `solutions` to 2, to decide whether there is a unique model.

Stack

The search control algorithm makes use of a database of pieces of information about the search space. There is a stack of "blanks" for use in this database. When the stack is exhausted, FINDER decides that the search job is too big to be handled in one hit, so it divides the remaining search space into disjoint subspaces and searches each separately. The stack is made up of two sections: the part used to hold information discovered by the preprocessor and the 'margin' used to hold that discovered during the search. By default, the stack is of maximal size (256 as FINDER is supplied) and the margin is of size 6. These are the numbers of batches of 1024 database entries.

The size of the stack may be set by means of the `stack` setting specification, and the size of the margin by means of `stack-margin`. Setting a small stack causes space division to happen often, with consequent repeated loss of information but with frequent refreshment of the database so that clutter is reduced. Setting a large stack reverses these effects. There is no general rule for predicting the optimum stack size: it varies from job to job. Generally, only the margin will need to be adjusted, unless you are experimenting with special effects or need to keep memory usage down.

There are symbolic constants for specifying stack sizes. They are defined in the header file Fdef.h. In increasing order, they are

```
itsy        % 1
bitsy       % 2
tiny        % 4
small       % 8
medium      % 16  (the default)
large       % 32
LARGE       % 64
huge        % 128
maximal     % as many as possible
```

You may give a numeral instead of a symbolic constant if you want one of the values in between, though there is no good reason to do so.

Test after

The setting specification `test-after` followed by a non-negative integer is for use when FINDER is an adjunct to another system such as OTTER (see §2.3.3 and §4.3 for this use of FINDER). It causes FINDER to stop generating models and start behaving as a tester after the specified number of clauses have been input.

Testing by variables or weight

During testing, clauses are treated in order. Normally the most significant factor in determining this order is the number of variables in the clause, with clause weight as the minor sort. Setting `test-by-vars` is the default for the toggle which causes this ordering to be imposed. Optionally, weight may be made more significant than number of variables. Naturally, this is communicated to FINDER by setting `test-by-weight`.

Time stamp

The setting `time-stamp` causes each model printed to be flagged with the time (in cpu seconds) since the start of the search. The default setting is `no-time-stamp` which turns off time stamping.

Time limit

A setting of `time-limit <n>` causes FINDER to jump out of the search after n seconds. Note that this refers to elapsed time, not cpu time. The dafault value is 0 (no limit).

Tops on and off

As explained in §3 refutations (constraints) which are too big or involve the most significant cells are reduced by decapitation. The setting `tops-off` or `tops-on` (default `tops-off`) enables or disables this action. Do not use it, as FINDER knows best about such things!

Verbosity

There are three verbosity levels: none, brief and full. There are also three aspects of a FINDER run on which information can be requested: the input job, the models found, and some overall statistics. The verbosity level for each of these three may be set independently of the others by means of the setting specification `verbosity` followed by a specification list of levels. The default setting is

```
verbosity {
    models: full
    job: none
    stats: brief
}
```

The level `none` is self-explanatory. In detail, the other two levels are as follows.

models Printing of solutions as they are found. The format may be either pretty or ugly.

- **brief** Each function is reported only when it or some more significant function changes, or when the cardinality of a sort changes.
- **full** Every function with print mode `full` is detailed every time a model is found, even if the table of values is the same as for the previous model.

job Printing of the current specifications.

- **brief** The sorts and functions only of the current job are printed out, with all of their specifications.
- **full** The sorts, functions, clauses and settings of the current job are printed out in that order, with all of their specifications.

stats Printing of a report on the completed job.

- **brief** At the end of the run, FINDER prints the number of models found and the total time taken.
- **full** In addition to the number of models, FINDER prints information about the numbers of subspaces, backtracks and constraints.

2.3 Non-standard input

2.3.1 Piping

It is possible to use output from FINDER as input to FINDER. For this purpose, the `ugly` output format and the command line switch `-f` or `-n` must be used.

An ugly printing of a model consists of the word `model` followed by a series of entries followed by a period. Each entry is either a function symbol followed by the value table for that function or the word `size` followed by a sort name followed by an integer denoting the cardinality of that sort. An ugly output file consists of the job, followed by the models found, followed by 'end'.

The word `ugly` among the settings causes output to be in the ugly format and also sets the verbosity levels to new defaults

```
verbosity {
    job: brief. models: brief. stats: none.
}
```

These may be over-ridden by subsequent specifications, but for purposes of piping the output to another FINDER job there is usually no need to do so.

The command line option `-f <filename>` is used to specify the file from which FINDER is to take input. The file is in addition to the standard input stream, not instead of it. If FINDER is given an input file in this way, it expects models to be piped into it from `stdin` and to execute its job on each one. It also expects a job specification (sorts and functions at least) to come from `stdin` before anything else happens, and will not proceed to read its file until it receives the `end` statement.

To illustrate with the usual example of the ordered semigroups of sizes up to 4, here is an input file called `SG.1`.

```
setting  ugly
sort     element {
             cardinality < 5
             cardinality > 1
         }
function *: element,element -> element {}
clause {
    a * b < a * c  ->  b < c.
    a * c < b * c  ->  a < b.
    a * (b * c) = (a * b) * c.
}
end
```

Now here is another one called `SG.2`.

```
function ~: element -> element { bijective cut print:none }
clause   a * ~(a) = ~(b) * b.
end
```

A rather inefficient way to generate ordered semigroups which admit a quasi-inverse is to pipe the output from FINDER on `SG.1` through FINDER with input file `SG.2`. The command line

```
% finder < SG.1 | finder -f SG.2
```

suffices for this. It is worth dwelling a little on what happens in this case.

```
% finder < SG.1
```

prints 436 models (out of 739 tested) in ugly format. It starts by printing the job description and then goes on to specify the sizes and the tables for the star.

```
sort {
   bool
   enum: false, true.

   int
   cardinality = 31

   element {
      cardinality > 1
      cardinality < 5
   }
}

function {
   *: element, element -> element {
      total
      scope: 1
      change-order +T
      print: full
   }
}

end
model
size int 31
```

```
size element 2
*  0 0 0 0
.
model
*  0 0 0 1
.
model
*  0 1 0 1
.
```

When this is piped into another FINDER along with `SG.2` the piped job description is read first, as far as the word `end`. Then the file `SG.2` is read, adding the invisible tilde function and the quasi-inverse clauses. Then, after the `end` of that input, the models are read from the pipe one by one, and in each case the second FINDER tries to add the tilde function in accordance with its specifications. Where it succeeds, it prints the model, in *its* output format of `pretty` since it does not inherit settings from the pipe. Where it fails, it prints nothing. The eventual result is to select from the 436 models which come along the pipe the 142 on which a quasi-inverse is definable.

It sometimes happens that an input file is created from which you usually want ordinary runs but sometimes want piped output. In that case the command line switch `-p` (for "pipe") will over-ride the setting specifications, causing ugly output with its default verbosities. To illustrate, the effect of removing the setting line from SG.1 and running

```
% finder < SG.1
```

is to cause the ordered semigroups to be printed in pretty form as usual. Then to pipe the output through the second FINDER and file the output, type

```
% finder -p < SG.1 | finder -f SG.2 > SGinv.out
```

There is no particular reason to use piping for the above problem, as the two input files could easily be combined into one. A typical case in which this cannot be done would be the problem of enumerating the ordered semigroups on which a quasi-inverse is *not* definable. That is, suppose we want to see not the 142 structures which do admit quasi-inverses but the 294 which do not. For this FINDER provides another command line switch, `-n <filename>`, which behaves just like `-f` as far as input is concerned, but gives as output exactly those models piped into it for which it fails to find a model of its larger theory. Hence

```
% finder -p < SG.1 | finder -n SG.2 > SGnoinv.out
```

will file exactly those models output from

```
% finder < SG.1
```

which did not get filed before. Before this negation facility was added to FINDER, its effect could only be secured by virtuoso use of `grep` and `awk` and shell script trickery. The real hackers can still do it their way, but the rest of us can now do it better.

2.3.2 Command line

There are four command line options. Options `-f`, `-n` and `-p` have been covered in §2.3.1. Option `-c` may be used to append material to what occurs in the input file. It is usual, and recommended, to place double quotation marks around such command line input, so that white space and other meaningful characters are treated by FINDER and not by the shell or by the operating system.

FINDER places white space around each piece of text input from the command line, so breaks between successive such items should not occur in the middle of words, but otherwise there is no significance to such breaks. The syntax of command line input is exactly the same as that of file input. It is treated as though it came just before the word `end`.

For a fairly pointless example, consider a FINDER file for enumeration of ordered semigroups of order 5:

```
sort element  cardinality = 5

function *: element,element -> element {}

clause {
    a < b, a * c > b * c  ->  false.
    a < b, c * a > c * b  ->  false.
    a * b * c = a * (b * c).
}

end
```

Suppose this is in a file called `OSG.in`. If we want to see the ordered semigroups of order 4 instead, without editing the file, we may type

```
% finder -c "sort element cardinality = 4" < OSG.in
```

If instead we wanted to send the ordered idempotent monoids of order 6 to an output file, we could type

```
% finder -c "sort element cardinality = 6" \
-c "function e: element.{}" \
-c "clause { x * e = x.  e * x = x.  x * x = x. }" \
< OSG.in > OIM.out
```

A more common use of command line input is to change the settings. For example, to view the runtime statistics without having to look at all the models:

```
% finder -c "setting verbosity " \
-c "{stats:full models:none}" < OSG.in
```

Note that the command line input is read last, so it can over-ride any contrary settings that may have been made in the file.

2.3.3 FINDER as TESTER

This section is expected to be most useful to anyone intending to incorporate FINDER into other programs, and therefore assumes some knowledge of C programming. This makes it in some ways more technical than the rest of these notes.

A facility is provided whereby FINDER can be called as a function from another C program. In order to do this, the parent program must link in all the FINDER object files, of course, with the exception of `Fmain.c` which contains the main function of the stand-alone version of FINDER.

In its use as a tester, the input configuration and the functionality of FINDER are a little different from normal. Firstly, the calling program is expected to be using the standard input and output streams for its own purposes, so FINDER cannot read from them. Hence before FINDER can be called, it is necessary to execute a statement

```
init_finder(path);
```

The parameter `path` is a character string containing the name of a file. The function `init_finder` returns a boolean value, which can be read as an integer 0 or 1 in the usual way. FINDER reads the input in file `path` very much as normal. The input may contain settings and other directives as described in the above sections. There need not be any clauses in the file, though there may be. FINDER will search for a model of whatever clauses are given, returning `true` when it finds one. If no clauses are given, of course, it “finds” the first

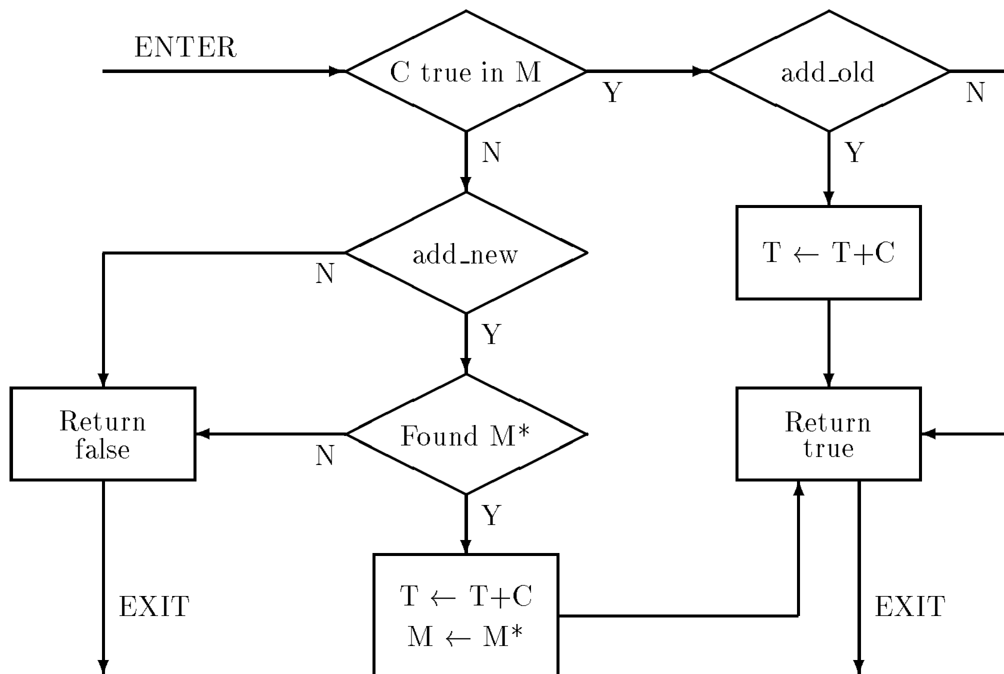
candidate model in its search space. If no model is found in the search space it has been instructed to search, it returns `false`. The model, if any, resides in memory until FINDER is called again, and the clauses, if any, also reside in memory. They will be referred to below as the current model and the current theory. They are not printed out by `init_finder`.

Subsequent calls to FINDER take the form

```
true_in_model(C, add_old, add_new);
```

where `C` is a character string representing a clause in the usual input format recognized by FINDER (see §2.1.1 and §2.2.3 above), while `add_old` and `add_new` are boolean flags (0 for false or 1 for true). The function `true_in_model` tests the clause `C` for truth in the current model. If it is true, `C` is added to the current theory if the parameter `add_old` is true, and `true_in_model` then returns `true`. If `C` is not true in the current model, and the parameter `add_new` is false, then it returns `false`. If the clause fails the test and `add_new` is true, then FINDER temporarily adds `C` as current clause to its current theory and searches for a model of the resulting enlarged theory. If it fails to find one it deletes current clause `C` from its theory and returns `false`. If it does find a model, it updates the current model to the one just found, confirms the current clause as part of its current theory and returns `true`.

Here is a flowchart of all that. M is the current model and T the current theory. M* is the new model, if such be found, of T plus the current clause C.



The upshot is that FINDER can be used to test arbitrary clauses for truth in a model of a given theory, and can also be caused to update the theory and model dynamically as things progress. Note that the theory grows monotonically: there is no way to delete clauses once they have been added.

If you are not concerned to keep fine control over when FINDER is to add clauses and when not, you may use the simpler call

```
is_true(C);
```

whose only parameter is the string containing the clause to be tested. FINDER treats this like a call to `true_in_model`, assuming that `add_old` and `add_new` are both true if only a few clauses have so far been tested, and both false if a lot have already been tested. The boundary between a few and a lot is by default 100 and may be set by means of the `setting` option `test-after`.

Several further functions are provided in order to make certain operations easier. A call of

```
theory_size()
```

simply returns the number of clauses in the current theory. You may, for instance, want to stop adding further clauses to the theory when it gets bigger than some limit. There are no parameters, and the value returned is an integer.

Secondly, the statements

```
fprint_the_model(f);
fprint_the_job(f);
```

cause FINDER to print respectively its current model and its current job. The model is printed in whatever format was specified in the input file. In either case, a verbosity level of `full` is assumed. The parameter is a pointer of type `FILE` and may be `stdout` or `stderr` if you wish. These functions do not return a value. To suppress the imposition of `full` verbosity, omit the '`_the`'.

Finally, the parent program may add more FINDER input, in the standard format, at any time. The usual way to do this is to read a file and transfer its contents line by line, though other devices are possible. The call is

```
finder_input(s);
```

where `s` is a null-terminated character string of any length. It is recommended that this function be used with care, unless you are sure that you really understand how FINDER handles its input, since the results may be a little unpredictable. This is particularly the case if you change the cardinality of

sorts, add new sorts or add new function symbols. The current model may get corrupted in the process. It would be more normal to use `finder_input` only to change settings, print modes and the like. If you have changed sorts or added functions you should refresh FINDER by adding

```
re_init_finder();      /* Note no parameter */
```

At any time after FINDER has been initialized and a current theory defined, it may simply be run to generate all models in its space as usual by means of the call

```
go(0);
```

Chapter 3

How it Works

Part of the design philosophy of FINDER is that the details of the actual search algorithm it uses are unimportant, so another algorithm could be implemented and slipped into its place with no significant change to FINDER's functionality. In order to understand some of the setting options and other features, however, a little knowledge of the inner workings of FINDER is necessary.

The outer algorithm is as follows.

```
FINDER:
   get job description
   lower DONE flag
   GO( first sort )
end FINDER


GO ( sort X ):
   for each possible cardinality c of X do
      set the cardinality of X to c
      if there is a next sort X+1 then
         GO ( X+1 )
      else
         if not DONE then
            set up index
            SEARCH( set of possible models )
         end if
      end if
   end for
end GO
```

There are some external (global) variables used by the various bits of FINDER to communicate with each other. A boolean flag `DONE` is used to indicate that

some termination condition has been recognized. For example, the maximum number of models may have been reached. There is also an index used to translate between the many data structures representing a model and the simple vector of values generated by the searcher. In addition, there is the job description which is quite complex, as it includes details of all the sorts, functions, clauses and settings. The flag that stops the search may be raised at any time during the search. Getting the job description and setting up the index are of course somewhat complex operations which are not detailed here.

Before the `SEARCH` algorithm can be presented, it is necessary to explain some terminology and in order to do this it is necessary to examine the structure of a tree search as solution to a constraint satisfaction problem.

Let us fix attention for the moment on a concrete problem. Suppose, as ever, that we are enumerating totally ordered semigroups. We have decided ahead of time that the domain consists of four elements called a, b, c and d and that they come in that (increasing) order. The postulates are

$$\begin{aligned} x \leq y &\Rightarrow x \star z \leq y \star z \\ x \leq y &\Rightarrow z \star x \leq z \star y \\ (x \star y) \star z &= x \star (y \star z) \end{aligned}$$

The corresponding FINDER input file reads

```
sort element { enum: a, b, c, d. }
function *: element,element -> element {}
clause {
    x < y,  x * z > y * z  ->  false.
    x < y,  z * x > z * y  ->  false.
    x * y * z  =  x * (y * z).
}
end
```

A typical model (one of the 386 that FINDER produces) is

```
Model 111

 * | a b c d
---+---------
 a | a a a a
 b | a a b b
 c | a b c d
 d | a b c d
```

This simple problem will serve to illustrate most of the features of FINDER's search algorithm.

The matrix for the semigroup operator consists of 16 cells corresponding to the pairs of elements. Each cell needs a value, and there are 4 possible values for each. Consequently, the search space consists of 4^{16} or 4,294,967,296 possible matrices. The idea is to generate a candidate matrix and test it for satisfaction of the postulates. If it passes the test it is printed out; if it fails it is changed somewhat to get rid of the detected badness, giving a new candidate which is tested in turn. The ways that postulates have been found to fail are stored in a database so that they can be avoided in future. In order to define the search tree, the cells are arranged in an order thus:

	a	b	c	d
a	1	2	3	4
b	5	6	7	8
c	9	10	11	12
d	13	14	15	16

The search controller works with this one-dimensional vector of cells, not with the more complex data structures of the model such as the two-dimensional array corresponding to the star operation. The mapping between the vector and the model is done via the index.

Since the domain is finite, each clause can be regarded as a set of ground clauses got by plugging in values for the variables. That is, the associativity postulate, for instance, amounts to 64 ground equations

$$\begin{aligned}
(a \star a) \star a &= a \star (a \star a) \\
(a \star a) \star b &= a \star (a \star b) \\
(a \star a) \star c &= a \star (a \star c) \\
(a \star a) \star d &= a \star (a \star d) \\
(a \star b) \star a &= a \star (b \star a) \\
(a \star b) \star b &= a \star (b \star b) \\
(a \star b) \star c &= a \star (b \star c)
\end{aligned}$$

and so forth. The search controller does not know about the star, so for its purposes we should express all this in terms of a binary relation between cells

and their values. It also does not know about a, b, c and d: it just thinks of the values as value #0, value #1 and so on. Let us write P_y^x where cell x has value y, so for example P_1^{13} means that cell 13 has value 1, which translates via the index into $d \star a = b$.

Now suppose (as actually happens in the sample case) FINDER constructs a candidate model which has these values in these cells:

	a	b	c	d
a		a		b
b				b
c				
d				b

It does not matter what values it has put in the other cells. Now one of the ground postulates is false of this structure. Specifically

$$(a \star d) \star d \neq a \star (d \star d)$$

Evidently this failure is not a global property of the whole candidate model, but only of the four cells involved. What we can tell the search controller as a result of discovering this badness is that

$$\neg P_0^2 \vee \neg P_1^4 \vee \neg P_1^8 \vee \neg P_1^{16}$$

That is to say, any failure of a postulate translates into a negative clause with as many (ground) literals as there are cells involved in the failure. Such a clause is what will here be called a (negative) constraint. The corresponding set of cell-value pairs is called a refutation, following terminology of Pritchard. A refutation of cardinality n is called an n-refutation. So the associativity failure in the above example is a 4-refutation. FINDER actually works directly with refutations, recording them in such a form that the relevant ones can be accessed and applied very rapidly, and does not explicitly formulate the corresponding clauses. The presentation in terms of clause-form constraints and inferences is given here only in order to lay out the logic of the search clearly.

There are also of course some understood positive clauses, saying that each cell has a value:

$$P_0^1 \vee P_1^1 \vee P_2^1 \vee P_3^1$$

and similarly for all the other cells. These are essentially contributed by a preprocessing routine which takes what FINDER knows to be impossible values out of the list of available ones for each cell. The tree traversal is driven by choosing exactly one literal from each positive clause (a value for each cell), assuming that literal as a new unit clause, resolving with it against the constraints to give smaller constraints and backtracking if ever the empty clause results. On backtracking, of course, all of the temporary smaller constraints must be deleted again. If one of the "smaller constraints" gets to be a unit clause, saying just that some value is not available for some cell, then it too can be used in a resolution step to reduce the number of disjuncts in one of the positive clauses. All the clauses considered on this view of the search logic are thus either positive ones or negative ones, and all of the primary inferences consist of applying a single literal to a clause of the opposite polarity in order to shorten it by one. These unit-resolution inferences are termed *constraint strengthening* and *space reduction* respectively in [18] where it is also pointed out that we may regard the search as tracing out a tableau-style logical proof.

For reasons of efficiency, n-refutations for small values of n are treated differently from those for large values. For a clause giving n-refutations for (approximately) $n < 4$ it is most efficient to generate all of the ground instances, and all of the refutations to which these can possibly give rise, in advance of the search. FINDER does this, along the way removing any refutation subsumed by one of its subsets, in a second preprocessing stage. Short refutations are thus written into its description of the search space, so that any candidate model generated is bound to satisfy them. If all clauses in a problem are of low weight, of course, no testing is necessary since all candidate models are actual models. It is not efficient to preprocess large refutations because there are so many of them and usually a rather small subset is sufficient to determine the models. 'Lazy constraint generation', whereby such refutations are discovered by generate-and-test methods, is therefore the technique of choice for them.

The `SEARCH` algorithm, to traverse a search space S, thus reads as follows. Note that S is empty if one of its cells has the null set of possible values.

```
SEARCH( space S ):
   initialize refutation database
   remove impossible values from S
   if S is nonempty then
      record small refutations in database
      GENERATE( S )
   end if
end SEARCH
```

This calls a recursive procedure `GENERATE` to traverse the tree. `GENERATE` calls a test procedure with each candidate vector in that part of search space `S`

which has values for certain cells already fixed but allows the values for the remaining cells to vary. In a somewhat simplified form, it reads

```
GENERATE( space S ):
   if every cell in S has a value then
      TEST ( vector of values )
   else
      choose some n-th cell for splitting
      for each possible value v of n-th cell do
         set n-th vector value to v
         remove incompatible values from S
         if S is nonempty then
            GENERATE ( S )
         end if
         un-set n-th vector value and re-set S
      end for
   end if
end GENERATE
```

This must be complicated a little to allow for the addition of newly discovered constraints to the refutation database. It also needs to keep checking the `DONE` flag to see whether it should stop. Moreover, it has to keep track of such details as how far to backtrack when a new refutation is encountered. These complications are not the present subject. It should, however, be noted that the choice of cell for splitting is made on the basis of the number of possible values remaining for it. A cell with few possible values is preferred to one with many, in order to reduce furcation of the search tree.

The `TEST` routine is fairly simple. It must first decode the vector put forward by the search controller, since this is just a list of the assigned values in their change order and has yet to be translated into testable data structures. If the candidate model passes the test, naturally, it should be accepted.

```
TEST( vector V ):
   use index to translate V into possible model
   for each clause C of testable weight do
      for each assignment of values to variables in C do
         compute values of all subformulas in C
         if antecedents of C true and consequents false then
            record cells used in evaluation as a refutation
            RETURN "bad"
         end if
      end for
   end for
   print out the model
   RETURN "good"
end TEST
```

If during the search the database of known refutations gets too big (where the meaning of 'too big' is stipulated by means of the stack size and stack margin setting in the input) then FINDER divides the search space into two or more disjoint subspaces and searches them separately, discarding the refutation database and repeating the entire preprocessing routine with each one. There is obvious inefficiency in thus repeating work, but the empirical evidence suggests that this is outweighed by the benefits of refreshing the database and keeping it compact.

It often happens that the search has to backtrack before the vector of values is complete, because the empty clause has been derived (that is, because there is no possible value left for some cell). In terms of the constraints, what has caused this? There must be constraints

$$\begin{array}{c} \neg P^i_0 \vee X_0 \\ \vdots \\ \neg P^i_n \vee X_n \end{array}$$

where cell i is the one with no remaining value, where there are $n+1$ initially possible values for it, and where each X_j is a (possibly empty) disjunction of negative literals each of which is false of the partial vector so far constructed. In that case, we can apply a rule of negative hyper-resolution using the positive clause

$$P^i_0 \vee \ldots \vee P^i_n$$

as the nucleus. This leaves a totally new constraint

$$X_0 \vee \ldots \vee X_n$$

which records at any rate one reason why the search had to backtrack at that point. This is a *secondary constraint*. It can be added to the refutation database in exactly the same way as are the primary ones corresponding to failures of input clauses. The result will be that never again will we backtrack for that reason. Because FINDER never[1] backtracks twice for the same reason, it is enormously more efficient than the penny-plain transferred refutations searcher. We should expect this, since the time complexity of constraint satisfaction problems is typically dominated by the backtracking behaviour. The sample results in §4.1 bear it out.

Although primary refutations are limited in size to the weights of the input clauses, secondary refutations are not. As a concrete example, during the search for 4-element ordered semigroups, FINDER comes across the primary 4-refutation noted above:

	0		1				1							1

[1] Well, hardly ever.

The least significant cell involved is #2, and this refutation prevents it from getting value #0 as long as the other three cells involved have value #1. But in virtue of the 'order' postulates we already know about the 2-refutations

	3											1		

	2											1		

	1	0												

Putting these four refutations together (using negative hyper-resolution, if we wish to think in terms of clauses) we get the secondary refutation

		0	1				1					1		1

This is of greater cardinality than any primary refutation could be in this particular job. Moreover, since secondary refutations can be used to generate further secondary refutations, very large refutations can build up quite quickly. The bigger a refutation is the less information it carries, for taking account of a refutation divides the number of possible vectors in the search space by a factor of $\frac{\mathcal{P}}{\mathcal{P}-1}$ where $\mathcal{P}$ is the product of the numbers of possible values in the cells involved. Evidently the effect on the size of the search space diminishes logarithmically with increasing cardinality of refutations. But the amount of work a refutation causes is roughly in direct proportion to its cardinality. Hence too many large refutations clog the database with uninformative junk which does little but make work. Fortunately, there is a device to mitigate this effect.

Faced with a refutation of excessive size, FINDER could simply refuse to store it. This, however, would be to reject too much of the information it carries. Instead, what it does is to diminish the refutation by removing from it the most significant cell-value pairs (where 'significant' means 'chosen high in the search tree'. Then the refutation is stored with a marker to say that it should be discarded when the search backs up to the point where one of the removed values is changed. Until then it is treated just like any normal refutation. In terms of the search construed as construction of a logical tableau, this is easily expressed as inserting the new clause not at the root of the tree but only at that of some more or less local branch. This treatment of over-large refutations can result in redundant work, for the same refutation may be discovered again after it has been discarded, but empirical evidence shows

that such extra work is outweighed by the efficiency gains from keeping out monstrous refutations. The logical correctness of the procedure is obvious.

Since for any given set of clauses the number of primary refutations is polynomial in the size of the domain of values, as is the time taken to find them all, and the time taken to apply them in constructing a candidate vector, provided no backtracking happens, the time taken per model found would also be polynomially bounded were it not for secondary refutations. Typical FINDER problems are NP complete on this measure, so the number and size of secondary refutations must eventually dominate in most cases. Hence all techniques designed to improve efficiency bring at most temporary relief. Over the thirteen years or so since the 'transferred refutations' algorithm was first implemented, many experiments have been made with devices for keeping the combinatorial explosion within bounds for a while. Most of these were rather inconclusive, but two ideas have proved really valuable. One is the cutting down of over-large refutations as outlined above. The other is the "divide and conquer" step triggered when the number of entries in the database of refutations exceeds a certain figure. Each of these involves a cost in repeated work which has to be balanced against the benefits gained. The settings for parameters to achieve this balance can only be determined experimentally in the case of any particular type of problem.

Chapter 4

Sample Applications

4.1 Problem solving

This section consists of sample problems suitable for FINDER, each with a FINDER input file representing the problem and with the results of running FINDER on that input. It is hoped that the examples given here will serve as both illustrations and templates.

There is no good benchmark set for search programs of FINDER's type. A ludicrous amount of the literature on constraint satisfaction problems is concerned almost exclusively with the Queens Problem (see §4.1.3). This is a highly atypical case, as the observations here attest. Moreover, its computational interest is slight, its mathematical and philosophical significance negligible, and its practical import zero.[1] Nonetheless, it is the standard case and so has to be treated in this section. Much more worthwhile problems arise in the theory of quasigroups and other group-like objects. These deserve a section to themselves, and so their discussion is postponed for a few pages. Perhaps the best benchmark I know, and certainly one which has been used intensively during the development of FINDER, is that of enumerating ordered semigroups. It finds a place here. The rest of the examples are included not so much to demonstrate FINDER's performance as to illustrate its use.

In seeking suitable problems for FINDER I found a useful source in the collections of puzzles sold commercially in newsagencies. One issue of such a magazine provided FINDER with several challenges, including the 'Squaring Up' puzzle detailed here. These puzzle-book problems are not deep, but they are of interest as the sort of logical exercise human beings enjoy.

[1]The other problem which gets discussed is graph colouring, which is a much better example because some real problems, in scheduling for instance, can be reduced to it. Graph colouring in general is still not an ideal testbed for algorithms, however, since the cases treated usually turn out to be trivial.

4.1.1 Jobs

This is a staple of automated reasoning. It is not difficult, but shows the general technique for representing such problems in FINDER's format. The version given here is lifted directly from [21] where it is given a sustained treatment as a sample problem for automated deduction.

Problem description

There are four people: Roberta, Thelma, Steve and Pete.
Among them, they hold eight different jobs.
Each holds exactly two jobs.
The jobs are: chef, guard, nurse, telephone operator, police officer, teacher, waiter and boxer.

The job of nurse is held by a male.
The husband of the chef is the telephone operator.
Roberta is not a boxer.
Pete has no education past 9th grade.
Roberta, the chef, and the police officer went golfing together.

Who holds which jobs?

FINDER input

```
sort {
    person
    enum: Roberta, Thelma, Steve, Pete.

    job
    enum: chef,  guard,  nurse,  telephone-operator,
          police-officer,  teacher,  waiter,  boxer.
}

function {
    male: person -> bool
    print: none

    female: person -> bool
    print: none

    job1: person -> job
    injective
```

```
    job2: person -> job
    injective

    is_a: person,job -> bool
    print: none

    husband: person -> person {
        partial
        injective
        print: none
        cut
    }

    educated: person -> bool {
        print: none
        cut
    }
}

clause {                    % These are common notions.
    male(x) = (female(x) = false).

    E!(husband(x)) -> female(x).
    E!(husband(x)) -> male(husband(x)).

    female(Roberta).
    female(Thelma).
    male(Steve).
    male(Pete).

    x is_a waiter  ->  male(x).

    x is_a nurse -> educated(x).
    x is_a police-officer -> educated(x).
    x is_a teacher -> educated(x).

    x is_a job1(x).
    x is_a job2(x).
    x is_a y  ->  y = job1(x), y = job2(x).
    job1(x) < job2(x).
    job1(x) = job2(y)  ->  false.
}
```

```
clause {                    % These are specific conditions.
    x is_a nurse  ->  male(x).
    x is_a chef  ->  husband(x) is_a telephone-operator.
    Roberta is_a boxer -> false.
    educated(Pete) -> false.
    Roberta is_a chef -> false.
    Roberta is_a police-officer -> false.
    x is_a chef, x is_a police-officer -> false.
}

end
```

Results and comments

The results are of no significance, but several comments are in order.

- Persons and jobs are of different types. This has several hidden consequences such as that no person can be a job. The enumeration order within each type is unimportant in this case.
- The representation given is by no means the most efficient. More function symbols are used than are necessary: only one sex need be primitive, as the other definable as its negation, and the relation `is_a` is redundant in principle. On the other hand, editorial freedom has been exercised in stipulating that degree of education is an all-or-nothing affair for this problem, rather than one involving arithmetic on grades.
- The order in which the function symbols come is significant. The ones which get trivially fixed come first.
- The `cut` specifications on `husband` and `educated` are to rule out multiple solutions differing only in marital or educational respects, in which we have no interest.
- The common notions should be fairly self-evident. To be male is, by stipulation, to fail to be female. It is necessary to the solution that waiters are male (female ones are “waitresses”). For no deep reason at all, one’s “first” job is the one earlier in the list of jobs than one’s “second”.
- The golf is irrelevant: the point of that clue is just that Roberta, the chef and the police officer are all different people.

4.1.2 Squaring Up

This neat puzzle is taken from page 39 of [5]. It is nontrivial for some search algorithms, though others, including that of FINDER, find it quite easy.

Problem description

The large square in the diagram is divided into 25 smaller squares, one of which is blank. The remaining 24 squares each contain one of the letters A to L or one of the numbers 1 to 12. From the clues given below, can you insert the letters and numbers into their correct squares?

1 There is only one number in horizontal row I; the letter two squares below it is the A.

2 Squares IVa and IVb contain respectively a letter and a number corresponding to that letter's alphabetical position. For instance, they could be A and 1, B and 2, and so on.

3 Square Va is the only corner square to contain a number; the number in Vc has twice its value.

4 There are three odd single-digit numbers in row III and one even number.

5 The blank square has a 4 immediately above it and an 8 immediately below, and letters to its right and left, one of which is the J.

6 Squares Ib and IIId contain respectively a letter and the number corresponding to that letter's alphabetical position.

7 Square IIb contains the 10 and square IVe the B.

8 The 7 is somewhere in column c but not in row I.

9 The F and the 11 are in adjacent squares in the same horizontal row, the former being to the right of the latter.

10 The L is two squares directly above the 3, but not in column e.

11 The I is two rows above the 9, but not in the same column.

12 The 1 is immediately above the 5 and the 6 immediately below the E.

13 The H is somewhere to the left of the K in the same horizontal row and the C is in a higher row than the G.

	a	b	c	d	e
I					
II					
III					
IV					
V					

FINDER input

```
   sort {
       row     enum: I, II, III, IV, V.
       column  enum: a, b, c, d, e.
       symbol  enum: [-],                                   % Blank
             '1', '2', '3', '4','5','6',
             '7', '8', '9', '10', '11', '12',               % Numbers
             'A', 'B', 'C', 'D','E','F',
             'G', 'H', 'I', 'J','K','L'.                    % Letters
   }

   function {
       even: symbol -> bool            print: none
       double: symbol -> symbol        { partial. print:none }
       num: symbol -> bool             print:none
       lett: symbol -> bool            print:none
       grid: row,column -> symbol      { bijective. no-priority }
       row-of: symbol -> row           { print:none. no-priority }
       column-of: symbol -> column     print:none
   }

   constant {                          % Skolem constants
       E: symbol                       print:none
       O1: symbol                      print:none
       O2: symbol                      print:none
       O3: symbol                      print:none
   }

   clause {
       E!(double([-])) -> false.
       double('1') = '2'. double('2') = '4'. double('3') = '6'.
       double('4') = '8'. double('5') = '10'.double('6') = '12'.
       E!(double(x)) -> x < '7'.
       num([-]) = false.
       x > [-] -> num(x) = (x < 'A').
       lett(x) = (x > '12').
       even(x) -> num(x).   even('1') -> false.
       even(x), even(x+1) -> false.
       num(x+2) -> even(x+1), even(x+2).
       row-of(grid(x,y)) = x.    column-of(grid(x,y)) = y.
       grid(row-of(x),column-of(x)) = x.
}
```

```
clause {
    num(grid(I,column-of('A'))).
    num(grid(I,x)) -> x = column-of('A').
    row-of('A') = III.

    num(grid(IV,b)).
    grid(IV,a) = grid(IV,b)+12.

    num(grid(V,a)).
    lett(grid(I,a)).
    lett(grid(I,e)).
    lett(grid(V,e)).
    num(grid(V,c)).
    grid(V,c) = double(grid(V,a)).

    even(E).              row-of(E) = III.
    even(O1) -> false.  row-of(O1) = III.
    even(O2) -> false.  row-of(O2) = III.
    even(O3) -> false.  row-of(O3) = III.
    num(O1). O1 < O2.  O2 < O3.  O3 < '11'.

    row-of('8') = row-of([-])+1.
    column-of('8') = column-of([-]).
    row-of('4')+1 = row-of([-]).
    column-of('4') = column-of([-]).
    column-of([-]) > a.
    column-of([-]) = x+1 -> lett(grid(row-of([-]),x)).
    lett(grid(row-of([-]),column-of([-])+1)).
    true -> column-of([-])+1 = column-of('J'),
            column-of([-]) = column-of('J')+1.
    row-of('J') = row-of([-]).

    num(grid(III,d)).
    grid(I,b) = grid(III,d)+12.

    grid(II,b) = '10'.
    grid(IV,e) = 'B'.

    column-of('7') = c.
    row-of('7') > I.

    row-of('F') = row-of('11').
    column-of('F') = column-of('11')+1.
```

```
        column-of('L') < e.
        row-of('3') = row-of('L')+2.
        column-of('L') = column-of('3').

        row-of('9') = row-of('I')+2.
        column-of('9') = column-of('I') -> false.

        row-of('5') = row-of('1')+1.
        column-of('5') = column-of('1').
        row-of('6') = row-of('E')+1.
        column-of('6') = column-of('E').

        row-of('H') = row-of('K').
        column-of('H') < column-of('K').
        row-of('C') < row-of('G').
}

end
```

Results and comments

This is the most complicated FINDER file in this section. Note that the straightforward numerals are not available for an enumerated sort other than `int` so the simple device of quotation has been used to get some new names. Together with the bracketed dash for the blank, this helps to show that the set of legal identifiers is quite generous. We have to tell FINDER what an even number in the range 1...12 is, and also what it is to double such a number. These bits of common knowledge are not parts of the clues as stated, however, so they have been separated out into their own clause section. The "grid" is the square of smaller squares to be filled in. This is one case where it is natural to use prefix rather than infix notation. Note that it is a bijection, even though it is dyadic. The extra functions for the row and column of a symbol are in principle redundant, though they make the clauses easier to read and assist a little with the search. The Skolem constants E (the even number) and O1...O3 (the three single-digit odd ones) are for clue 4.

The solution is generated in a second or so on a SPARC-2, though this time should not be regarded as more than a very rough result since it depends heavily on the details of the presentation. With some fine tuning of functions and clauses it may be reduced. If less care is taken over defining 'double' and the like then the search can take much longer. Note that FINDER proves there is only one solution within the search space. Hence it is a useful tool for the composer of puzzles who is concerned that there should be exactly one solution and no redundant clues.

4.1.3 Queens

Problem description

This problem comes in two versions. One problem is to find an arrangement of n queens on an $n \times n$ chessboard such that no queen attacks any other along rank, file or diagonal. The second form of the problem is to enumerate all the ways of so positioning the n queens. The FINDER input below is for this second form of the problem.[2]

The positioning of the queens is represented as a function assigning to each rank a file where the queen on that rank is placed. By making this function bijective, we assure that no two queens stand a rook's move apart. Clauses are then needed to stipulate that each queen covers its two diagonals.

FINDER input

```
sort {
    rank  cardinality = 10
    file  cardinality = 10
}

function queen: rank -> file
    bijective

clause {      % Note: r is a rank, x is an integer.
    E!(r+x),
    queen(r)+x = queen(r+x)  ->  x = 0.

    E!(r+x),
    queen(r) = queen(r+x)+x  ->  x = 0.
}

setting {
    primary
    verbosity models: none
}

end
```

[2]There is no clear reason to apply any search algorithm to the first form of the problem, since analytical solutions for all n have been known since early this century, and in any case methods such as simulated annealing can generate solutions for *millions* of queens in short order whereas search methods cannot. See [19] for these observations.

Results and comments

The first thing to note on running FINDER with this input is that in generating the 92 solutions it does not test anything else. This is because all the constraints are binary ones and so get incorporated before the tree search starts. Problems in which all of the primary constraints are binary are a little unusual, but they do arise. What makes the Queens decidedly peculiar is that processing secondary constraints actually slows down the search. With the above settings, FINDER generates all solutions to the problem of 12 queens in 66 seconds on a SPARC-2. With the word `primary` commented out, it take 97 seconds. Another curiosity is that FINDER 2 was faster on this problem than FINDER 3, although the latter is more efficient in terms of the size of its search tree. At least in small cases of the Queens Problem, the decrease in the number of branches (about a factor of 2 in these small cases) is insufficient to compensate for the greater time (about a factor of 4) taken to explore each branch.

Here are some results for the problem of n queens, giving timings with the default settings and also with the setting `primary`.

number of queens	number of solutions	FINDER 3 time (default)	FINDER 3 time (primary)	FINDER 2 time (primary)
8	92	0.4	0.4	0.2
9	352	1.1	0.9	0.5
10	724	4.0	3.0	1.6
11	2680	18.5	12.8	6.6
12	14200	97.0	66.4	34.8
13	73712	539.8	354.3	189.3

An interesting feature is that the time taken per solution found does not appear to depend much on the size of the problem, provided secondary refutations are suppressed. With a large number of queens, in fact, there are so many solutions, packed so densely into the search space, that the time taken to generate them is dominated by the number of solutions while the time taken to find just one solution remains small. This is another reason why the Queens Problem is unsuitable as a benchmark. Nonetheless, it is a staple of the literature, so there is some interest in observing FINDER's approach to it.

4.1.4 Heap Arithmetic

Nobody in their right mind would really look for a solution to this problem by tree search methods. Any term-rewriting system should be able to solve it almost instantly, and generally any quasi-deductive system capable of equational reasoning should find it easy. It is included here to illustrate some features of FINDER, and because it is like a dog walking on its hind legs.

Problem description

Members of some primitive tribes are supposed to use a counting system something like 'One, two, three, Heap' where the characteristic of Heap is that it is its own successor (heap many buffalo and another buffalo add up to heap many buffalo). It seems likely that this tale tells us more about anthropology and the vagaries of radical translation than it does about any conceptual lack on the part of the "primitives", but that aside the notion of a *terminus a quo* for the natural number system is one that makes a kind of sense.

Except that Heap is its own successor, all is as Dedekind and Peano said it was. Natural addition, multiplication, exponentiation and the like can be given their usual recursive definitions. This contrasts with other finitizations such as modular arithmetic, where in general exponentiation is not well defined. Also naturally defined in Heap arithmetic but not in modular arithmetic is the numerical total order as the ancestral of the relation every number bears to itself and its successor. The problem is to derive the table for exponentiation from the recursive axioms.

FINDER input

```
sort number
enum: zero,one,two,three,four,five,six,seven,eight,Heap.


function {
    succ: number -> number {}

    plus: number,number -> number  {}

    times: number,number -> number {}

    power: number,number -> number {}
}
```

```
clause {
    succ(Heap) = Heap.
    x < Heap  ->  x < succ(x).
    x < y  ->  succ(x) = y, succ(x) < y.

    x plus zero = x.
    x plus succ(y) = succ(x plus y).

    x times zero = zero.
    x times succ(y) = (x times y) plus x.

    x power zero = one.
    x power succ(y) = (x power y) times x.
}


setting {
    pre-test: 4
    verbosity stats: full
}

end of input.
```

Results and comments

The clauses for the successor function say with a minimum of fuss that the successor of x is the next number after x, except in the special case of Heap. The other clauses are precisely the standard recursive axioms. FINDER cannot carry out recursive inferences, so it must fill in the tables for the four functions (that is, $3n^2 + n$ values, with n choices for each, for Heap arithmetic with n numbers) guided simply by the constraints. The search space naïvely consists of 10^{310} possible structures, though of course the constraints leave only one possibility.

The `pre-test` setting is important. The recursion axioms are of weight 4, and so would not normally be pre-processed. By means of the setting we force FINDER to discover all of the primary constraints before starting to search. If we left `pre-test` set to its default value of 3 the program would have to find the required constraints by generate-and-test. This would slow it down dramatically. In fact, since the recursive definition builds up the tables from the zero cases, it pays to have the test try low values for variables before high ones—the opposite of its normal strategy. This may be achieved by means

of the `reverse-test` setting. With that setting but without any `pre-test` specification, FINDER takes about 10 seconds to solve the problem for the case of Heap = 6 and 25 seconds for Heap = 7. Here, then, is an example where lazy constraint generation is a bad idea.

It is worth examining the statistics printed at the end of the run as a result of the input specified above:

```
Subspaces prepared:     1
Subspaces preprocessed: 1
Subspaces searched:     1
Solutions found:        1
Back subsumptions:      268
   Back subs by units:  268
Pre-test refutations:   10896
Total database entries: 10896

Branching:  1: 374

Preprocess time:        3.10
Search time:            0.18
```

The clauses gave rise to 10896 ground instances of the constraints, plus 268 more which turned out to be back subsumed by unit clauses (that is, were detected to contain impossible values). The preprocessing of these constraints took a little over 3 seconds on a SPARC-1. When the search started, naturally, there were always cells capable of only one value, so no nontrivial branching occurred at all, no secondary constraints were added to the database and no backtracks were recorded. Since the program went straight to the solution, only a fraction of a second was required for the 'search'.

4.1.5 Ordered semigroups

This problem has been used as an example in several places in the sections above, since it is so simple and clear. It forms a good benchmark for backtracking search programs, since it scales easily to any size and is fairly well immune to trickery. FINDER's sister program MaGIC is geared exclusively to enumerating ordered algebraic structures rather closely related to these, so the search algorithm they share was developed with such applications in mind.

Problem description

Enumerate the totally ordered semigroups with n elements. Only the number of such algebras need be printed when this problem is used as a benchmark. The postulates are as follows.

$$\begin{aligned} x \leq y &\Rightarrow x \star z \leq y \star z \\ x \leq y &\Rightarrow z \star x \leq z \star y \\ x \star (y \star z) &= (x \star y) \star z \end{aligned}$$

FINDER input

```
sort element
cardinality = 5

function *: element,element -> element {}

clause {
    x * z < y * z  ->  x < y.
    z * x < z * y  ->  x < y.
    (x * y) * z = x * (y * z).
}

setting {
    verbosity {
        models: none
        stats: full
    }
}

end
```

Results and comments

The cardinality of 5 is for illustration only. Here are the results of running FINDER on the above input with different choices of cardinality. The times given were obtained on a SPARC-2 with a 40 MHz clock and plenty of memory.

number of elements	number of solutions	bad ones tested	other backtracks	time in seconds	FINDER 2 time
1	1	0	0		
2	6	0	0		
3	44	32	1	0.1	0.1
4	386	278	22	0.6	0.4
5	3852	1333	331	9.2	6.7
6	42640	6895	3519	169.3	120.1
7	516791	66359	60601	2963.1	2674.8

FINDER splits the order 6 search space into 3 subspaces and the order 7 one into 31; in all smaller cases it searches only one subspace. Note that once again FINDER 2 was faster, at least on small cases of this problem, than FINDER 3. This seems to be because the search ratio (backtracks/models) is quite low, so that efficiency savings are hard to come by. Note that as the problems increase in size so FINDER 3 begins to catch up.

It will be evident that each successively larger problem is an order of magnitude harder than its predecessor. Hence for any model generator, some case of this problem will be on the performance limit, making it a worthwhile benchmark.

4.2 Quasigroup problems

A quasigroup consists of a set on which is defined a dyadic operation. The operation is cancellative in both left and right argument places, or, in FINDER's jargon, is both left bijective and right bijective. This means that the Cayley matrix for the quasigroup operation is a Latin square: each row and each column is a permutation of the elements. Quasigroups give rise to some interesting and difficult problems for search programs like FINDER. In this section, we briefly examine two such problems.

The simpler of the two is to find models of the equation $(ba.b)b = a$. This equation recieves a sustained investigation in [2] where several open problems are noted. The existence of idempotent models of given sizes is particularly in doubt. Prior to 1990, it was known that there exist idempotent models of every finite cardinality with the known exceptions of 2, 3, 4 and 6 and with 56 possible exceptions of which the largest was 174 and the smallest 9, 10, 12, 13, 14, 15 and 16. The order 9 problem was solved (negatively) by Jian Zhang in 1991 and the order 10 and 12 problems (also negatively) by Masayuki Fujita in 1992. The order 13, 14 and 15 problems were solved (negatively) by Mark Stickel in 1992–3. All of these results have been confirmed by FINDER and independently by Mark Wallace using the constraint logic programming system ECLIPSE. At the time of writing, the order 16 problem is still open, though maybe not for long.

Here is the FINDER input for the sample problem (QG5 of [7] and [18]).

```
sort int cardinality = 10

setting verbosity stats: full

function {
    cycle: int -> int print: none
    o: int,int -> int {
        row-bijective
        column-bijective
        change-order -rb
    }
}

clause {
    cycle(0) = 0.
    cycle(x)  ->  E!(x+1).
    true -> cycle(x), cycle(x+1).
    cycle(x)  ->  cycle(x+1)+1 = cycle(x).
```

```
        E!(y+1), x < y, cycle(x+1) < cycle(y+1)  ->  cycle(x).
        x  ->  x o 0 < x, cycle(x).
        cycle(x)  ->  x o 0 = x+1.

        x o x = x.
        y o x o y o y = x.     % (1)
        y o (x o y) o y = x.   % (2)
        y o (x o y o y) = x.   % (3)
}

end
```

There are several points to explain here. The function `cycle` is there to help avoid searching too many isomorphic subspaces. The clauses for it force the condition that in the column $x \circ 0$ the cycles of the permutation occupy contiguous sections of the numbering and occur in monotone decreasing order of length. The dot, of course, is the quasigroup operation. In addition to equation (1) which directly represents the defining condition for the problem, equations (2) and (3) have been added. Each is equivalent to equation (1) in the set of quasigroups, and together they help by adding more primary constraints. The change order `-rb` makes things happen a little faster, though it is not essential.

The order 10 problem shown above is rather trivial for FINDER 3 though it was beyond the reach of FINDER 2. This is a striking example of the greater efficiency of the new search algorithm. Several further equations have been investigated using FINDER and other general-purpose search programs. They include

$$\begin{aligned}
(x \circ y) \circ (y \circ x) &= x \\
(x \circ y) \circ (y \circ x) &= y \\
(x \circ y) \circ y &= x \circ (x \circ y) \\
((x \circ y) \circ x) \circ y &= x \\
(x \circ (x \circ y)) \circ y &= x
\end{aligned}$$

In all cases except the last, either FINDER or one of the other programs mentioned above has obtained solutions to open problems. Details are in [18].

The second quasigroup example is related, though slightly different in style. Two quasigroups given by operations $\circ$ and $\star$ on the same set of elements are said to be orthogonal if $a \circ b = x \circ y$ and $a \star b = x \star y$ cannot both happen unless $a = x$ and $b = y$. In other words, orthogonality is equivalent to the existence of 'row' and 'column' functions R and C such that for all elements a and b

$$\begin{aligned}
R(a,b) \circ C(a,b) &= a \\
R(a,b) \star C(a,b) &= b
\end{aligned}$$

or equivalently, for all elements a and b

$$R(a \circ b,\ a \star b) = a$$
$$C(a \circ b,\ a \star b) = b$$

It is easily seen that in such a case R and C also determine an orthogonal pair of quasigroups over the same base set.

A quasigroup operation $\circ$ over a set S gives rise to six quasigroup operations over S known as its conjugates. They may be denoted by subscripting as follows. These equations are all equivalent to $a \circ b = c$:

$$\begin{aligned} a \circ_{123} b &= c \\ a \circ_{132} c &= b \\ b \circ_{213} a &= c \\ b \circ_{231} c &= a \\ c \circ_{312} a &= b \\ c \circ_{321} b &= a \end{aligned}$$

A quasigroup which is orthogonal to its (ijk)-conjugate is said to yield an (ijk)-conjugate orthogonal Latin square or (ijk)-COLS. If idempotent, it is an (ijk)-COILS. One of order v is an (ijk)-COLS(v) or (ijk)-COILS(v). Interest attaches to the question of whether there exists an (ijk)-COILS(v) in various cases (see [18] for details).

FINDER has been used, for example, to search for (321)-COILS(v). In setting up this problem, we should first represent it in terms of the row and column functions above:

$$\begin{aligned} R(a,b) \circ C(a,b) &= a \\ R(a,b) \star C(a,b) &= b \\ R(a \circ b,\ a \star b) &= a \\ C(a \circ b,\ a \star b) &= b \\ a \star b = c &\Leftrightarrow c \circ b = a \end{aligned}$$

Applying the last of these equations to the second, we get

$$b \circ C(a,b) = R(a,b)$$

hence by substituting in the first equation

$$(b \circ C(a,b)) \circ C(a,b) = a$$

This equation turns out to be sufficient to force all of the rest to hold. A useful equivalent is

$$C((x \circ y \circ y), x) = y.$$

so we may enter the following FINDER input file to generate, for example, the (321)-COILS(7).

```
sort int cardinality = 7

setting verbosity stats: full

function {
    cycle: int -> int print: none
    o: int,int -> int {
        row-bijective
        column-bijective
        no-priority
    }
    |: int,int -> int {
        row-bijective
        column-bijective
        print: none
   }
}

clause {
    cycle(0) = 0.
    cycle(x)  ->  E!(x+1).
    true -> cycle(x), cycle(x+1).
    cycle(x)  ->  cycle(x+1)+1 = cycle(x).
    E!(y+1), x < y, cycle(x+1) < cycle(y+1)  ->  cycle(x).
    x  ->  x o 0 < x, cycle(x).
    cycle(x)  ->  x o 0 = x+1.

    x o x = x.
    x | x = x.
    y o (x | y) o (x | y) = x.
    (x o y o y) | x = y.
}

end
```

4.3 FINDER in deduction

Some of the more interesting applications of FINDER have been in the field of automated deduction. It is straightforward to use a model generator for what Bledsoe [3] calls 'reasonableness checking', and indeed this technique is extremely powerful in combination with many types of theorem prover. See [1] and [20] for discussions of how useful it can be. Less obvious, but also of interest, are some applications to standard resolution-based theorem proving systems which have been the subject of recent experiments with FINDER (see [16]). The present section is an outline of those experiments and some of the results.

The idea of semantic resolution has been around almost as long as resolution itself. Briefly, the biggest problem in searching for proofs of difficult theorems by unification and resolution is that there are just too many ways for the rules of inference to apply, so the thread of proof gets lost amid a tangle of irrelevancies. Techniques for cutting down the number of inferences made, without compromising completeness, are therefore of great value in automated proof discovery. Now suppose we have a model M and can easily tell whether a given clause is true in it. A simple theorem states that if there is a derivation of the empty clause from a set of clauses by unification, resolution and factoring then there is such a derivation in which no two clauses both true in M are ever resolved together. Resolution guided by a model in accordance with this fact is called semantic resolution.[3] Evidently, a model generating program such as FINDER can be used to generate structures appropriate to resolution proof searches by treating subsets of the clauses derived as the theories to be satisfied. The use of FINDER as a tester was devised for just this purpose.

The theorem prover used in the experiments was OTTER, a high performance resolution based prover developed by Bill McCune, building on earlier work by Lusk, Overbeek and others, at Argonne National Laboratory. OTTER is highly engineered, coded in C and has been in the public domain for some time. It is widely regarded as the leading prover in its classs, especially for problems with an algebraic flavour as opposed, for instance, to problems in set theory or in analysis. Because of its strength in the area of algebra, it seems quite appropriate as a mate for FINDER, and because it is one of the fastest provers around, any improvement in its abilities due to FINDER counts as a genuine advance in the field.

In simple terms, the algorithm used by OTTER is a straightforward implementation of the set of support method. The clauses are divided into a *usable list* and a *set of support* (that is, into two disjoint subsets). At the start, the

[3] Sometimes this is called *model resolution*, semantic resolution being what is here regarded as semantic hyper-resolution whereby in each inference one parent (the nucleus) is true in the guiding model while all other parents and the hyper-resolvant are false.

set of support must be non-empty while the usable list may optionally have clauses in it or not. No two clauses from the initial usable list are ever resolved together. When FINDER is used along with OTTER, it is usually best to put all input clauses in the set of support initially.

The main algorithm is then a loop:

```
while the set of support is not empty do
    pick a given clause g from the set of support
    move g to the usable list
    generate all resolvents from the usable list
        which use g as one of their parents
    for each such resolvent r do
        if r is the empty clause then
            report success
            stop
        else if r is not subsumed by another clause then
            add r to the set of support
        end if
    end for
end while
```

There are many optional settings and choices of rules of inference such as hyper-resolution, unit-resulting resolution, paramodulation and demodulation. OTTER also incorporates a sophisticated indexing technique allowing subsumption tests and the like to be carried out reasonably quickly. The great strength of OTTER is that it remembers all the (non-redundant) clauses it has generated, thus avoiding much repetition of labour. Its weaknesses are firstly that its inference speed is low, as it performs only tens of unification and resolution steps per second instead of the thousands done by some other provers, and that it runs blindly, making all possible inferences from each clause without regard to their relevance to the problem in hand. The virtue of using FINDER in conjunction with OTTER is that semantic information, about the meaning of clauses in relation to the particular goal, can be brought to bear. The effects are sometimes very good, sometimes modestly worthwhile and sometimes worse than useless. This should be expected in automated deduction, where there are no magic bullets and where no method suits every problem.

Naturally enough, OTTER's performance is easiest to improve where it is worst to begin with. Its authors point out that it is not the theorem prover of choice for propositional problems or for problems with many non-Horn clauses. One problem which combines these properties is the pigeonhole problem of showing that $n+1$ pigeons will not fit into n holes. This can be expressed in propositional form. OTTER was given a simple input file for the 5 pigeons

problem, using none of its special settings but simply asking it to find a proof by binary resolution. After one hour (!) on a SPARC-1 it was nowhere near a solution. A simple FINDER input file was prepared, containing only the information that each of the propositional variables was a function symbol of type `bool`. Using information from FINDER, the hybrid system was able to produce a proof in 75 seconds. [4]

For these theorem-proving purposes, FINDER is used as a tester, as detailed in §2.3.3. Refer to that section for an account. The basic algorithm of OTTER is extended to incorporate FINDER, giving a hybrid system which has been implemented as SCOTT (Semantically Constrained Otter).

```
init_finder (Finderfile)
while the set of support is not empty do
    pick a given clause g from the set of support
    if is_true (g) then
        mark g as ‘safe’
    else
        mark g as ‘unsafe’
    end if
    move g to the usable list
    generate all resolvents from the usable list
        which use g as one of their parents
        and have at least one ‘unsafe’ parent
    for each such resolvent r do
        if r is the empty clause then
            report success
            stop
        else if r is not subsumed by another clause then
            add r to the set of support
        end if
    end for
end while
```

The number of clauses to be treated by attempting to generate models, before FINDER gives up the search for a better theory and starts behaving just as a tester, is fixed with the `setting test-after` option in the FINDER input file. That file is called `Finderfile` in the above schematic algorithm. Also in FINDER's input file must be the function symbols used by OTTER in the proof, and there may be clauses helping to direct FINDER to a good model.

[4] Lest it be thought that this is an outstanding success for the method, it should be noted that by making full use of the set of support strategy OTTER can solve the problem in around 90 seconds, actually taking fewer given clauses than SCOTT. FINDER alone solves the problem in about 0.1 second. Indeed, it reproduces this solution during one of the calls of `is_true` from SCOTT, so the loss of semantic information by SCOTT is still quite serious.

This gives us a point at which domain-specific expert knowledge can be applied to help with the proof search without actually cheating by dictating lemmas to the prover.

For instance, one set of problems on which OTTER is very good comes from the pure implicational fragment of propositional logic. Łukasiewicz gave a single axiom for the classical pure implication calculus:

$$((p \rightarrow q) \rightarrow r) \rightarrow ((r \rightarrow p) \rightarrow (s \rightarrow p))$$

He thus provided some serious theorem-proving work in deriving a more usual set of axioms from this one using detachment (modus ponens) only. The following four are well known to suffice.

$$\begin{array}{l} p \rightarrow p \\ p \rightarrow (q \rightarrow p) \\ ((p \rightarrow q) \rightarrow p) \rightarrow p \\ (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r)) \end{array}$$

Of these, the first three are not too difficult to derive, but the fourth is a genuine problem. OTTER can derive theorems by condensed detachment (detachment with unification from axioms instead of axiom schemes) by using hyper-resolution instead of plain binary resolution. Unaided, it proves the fourth theorem in around 160 minutes on a SPARC-1, deriving some 6.7 million formulas, most of which are subsumed by things derived earlier. Just over 20,000 are kept (added to the set of support). With FINDER to help, given a little knowledge of what models for implicational logics look like, the time is reduced to under an hour, the clauses generated to 1.5 million and the kept clauses to under 14,000. That is, the improvement is roughly by a factor of two. This may not seem dramatic, but recall that OTTER's performance was previously the best ever recorded by any theorem prover.[5] A similar degree of improvement is found on a range of similar problems (for example, on the other three Łukasiewicz problems and on related problems arising in many-valued logic). On at least one problem reported in [16], SCOTT was over 3000 times faster than the most 'naturally' configured OTTER and still over 50 times faster when OTTER's settings were hand-tuned to a high degree.

While the completeness of SCOTT for binary resolution is a simple consequence of well known results, its completeness for condensed detachment problems is an open question. It is clear that SCOTT is incomplete for hyper-resolution in general.

There are other ways of using semantic information gleaned from small models which do not threaten completeness. For instance, it can be used as

[5]This is no longer the case, as the prover MGTP-N [9] now appears to hold the records for this class of problems. OTTER is still very competitive, however.

a heuristic to guide the choice of a given clause. If we worked with several small models each of which makes most of the derived clauses true and the goal false, then we could prefer given clauses which are false in as many of the models as possible, on the grounds that they have better chances of implying the goal. Preliminary experiments using just one model are rather encouraging. OTTER selects 'given' clauses by weight, lightest first. SCOTT can be instructed to test each 'kept' clause in its current model and to add something to the weight of each true clause. The effect of this is to cause false clauses to be given in preference to true ones. The value of the 'something' to be added has at present to be set by hand. This 'false preference strategy' has sometimes given improvements of around two orders of magnitude in time taken to solve hard problems, but performance depends quite significantly on how much extra weight is added to the true clauses. Research into this phenomenon is continuing.

In Conclusion

Although FINDER 3.0 is a large advance on the earlier versions, it must still be regarded as little more than a prototype. It is intended that numerous additions will be made before it is released again. Most obviously

- Function symbols of arbitrary arity must be allowed. The present restriction to dyadic functions is a nonsense.
- There should be a facility to define functions in terms of each other. Definitions need not be merely abbreviatory: such functions of argument x as 'the sum of the $f(x, y)$ such that $P(x, y)$' are easy to work with and should be made available. It should also be possible to define such things as the transitive closure of a dyadic function, and the inverse of a bijection.
- Many problems would be easier to represent if containment and other Boolean relations were allowed among the sorts. This should be incorporated in future releases of FINDER.

None of these improvements is especially difficult to code, but none is exactly trivial either.

Much more interesting extensions to FINDER's capabilities come in the form of several research projects. In outline these are as follows.

- The current FINDER and its future releases will parallelize very neatly. The program already has the capability to divide its search space into several disjoint subspaces, and there is no reason why these should not be searched simultaneously by parallel processes. Any process which finishes its subspace can ask for another, causing some active process to divide its task into new fragments, thus keeping the load well balanced without any particular programming effort. A parallel version of FINDER is under development and should be available soon.
- FINDER is limited to discrete problems, where each variable takes one of a finite list of possible values. An interesting extension would be to

allow floating point values instead, taking FINDER in the direction of scientific problems. The values would have to be discretized, of course, but various techniques for numerical solution by successive approximation seem fairly obvious. There is a possibility here of a new kind of solution to a new kind of problem, one involving perhaps hundreds of variables related in complicated ways by thousands of constraints. Many problems in the social and biological sciences would seem to be like this. Searching for stable states of an economic system or of a mix of species in a habitat, or for near-optimal production schedules in a factory, or synchronization patterns of traffic lights in a city centre come to mind immediately as examples. These problems cannot usefully be formulated for FINDER as it is, but they are essentially constraint satisfaction problems in many variables which do not require solutions accurate to ten significant figures.

- As these notes have been at pains to stress, the search algorithm used by FINDER is not supposed to be the best of which mankind is capable. Entirely different methods, such as arc consistency and path consistency algorithms, seem to be preferable to backtracking in the case of binary constraints. It would be quite feasible, and interesting, to use them to supplement of FINDER's current method. Also interesting would be a high-performance implementation of Pritchard's SCD algorithm (see [13] for a brief account). This performs as tree search, as does FINDER's algorithm, but one whose branch points correspond not to the choice of values for a cell but to the choice of a refutation from the list of those known. It can easily accommodate secondary constraints (see §3) though to the best of my knowledge it has never been implemented in such a way as to take advantage of them. There is fundamental research of an empirical sort to be done here. The best of all worlds might be to give FINDER many different search modules, using different algorithms, and either choose the one that looks best for a given problem or let them all run, perhaps in parallel, in competition or in co-operation with each other.

- The application of FINDER to theorem proving outlined in §4.3 is only one of many such potential uses for the program. The earlier program KRIPKE, as described in [20], made heavy and essential use of models of the kind produced by FINDER to achieve worthwhile results in automated deduction for non-classical logic. Much more work along similar lines remains to be done. The research on semantic resolution using FINDER or a similar program to generate the models dynamically is also capable of considerable extension. It would be interesting to see model generators applied to planning systems and the like, and of course some real life case studies, perhaps from timetabling or other scheduling

problem areas, would be of great value.

The future, then, is non-empty. For the present, its users are urged to treat FINDER as experimental software. Copying and modification of the program, within the copyright provisions, are not only allowed but encouraged. New applications, suggestions for extension or improvement, experimental results pertaining to the present program and constructive criticisms are always welcome, and of course if you feel like telling all your friends how absolutely splendid FINDER is, how its author ought to have tenure and so forth, please feel no inhibitions.

Bibliography

[1] M. Ballantyne & W. Bledsoe,
On Generating and Using Examples in Proof Discovery,
Machine Intelligence, 10 (1982), pp. 3–39.

[2] F. Bennett,
Quasigroup Identities and Mendelsohn Designs,
Canadian Journal of Mathematics, 41 (1989), pp. 341–368.

[3] W. Bledsoe & R. Hodges,
A Survey of Automated Deduction,
H.E. Shrobe (ed), **Exploring Artificial Intelligence**,
Morgan Kaufmann, San Mateo CA, 1988

[4] R. Caferra & N. Zabel,
Extending Resolution for Model Construction,
Proceedings of Logics in AI (European Workshop JELIA '90),
Springer-Verlag 1991. LNAI 478 (J. van Eijck Ed.), pp.153-169.

[5] A. Duncan & A. Gresty,
Logic Problems 69,
Quality Puzzle Magazines, British European Associated Publishers,
London, 1991.

[6] Esprit 3125—Medlar
Mechanizing Deduction in the Logics of Practical Reasoning
Progress Report PPR1, Imperial College, London, 1991.

[7] M. Fujita, J. Slaney & F. Bennett,
Automatic Generation of Some Results in Finite Algebra,
Proceedings of the 13th International Conference on Artificial Intelligence, 1993, pp. 52–57.

[8] R. Haralick & G. Elliott,
Increasing Tree Search Efficiency for Constraint Satisfaction Problems,
Artificial Intelligence, 14 (1980), pp. 263–313.

[9] R. Hasegawa, M. Koshimura & H. Fujita,
MGTP: A Parallel Theorem Prover Based on Lazy Model Generation,
Proceedings of the 11th International Conference on Automated Deduction, 1992, pp.776–80.

[10] R. James, M. Newman & E. O'Brien,
The Groups of Order 128,
Journal of Algebra, 129 (1990), pp. 136–158.

[11] W. McCune,
OTTER 2.0 User's Guide,
Technical Report ANL-90/9, Argonne National Laboratory, Argonne Illinois, 1990.

[12] B. Nadel,
Constraint Satisfaction Algorithms,
Computational Intelligence, 5 (1989), pp. 188–224.

[13] P. Pritchard,
Algorithms for Finding Matrix Models of Propositional Calculi,
Journal of Automated Reasoning, 7 (1991), pp. 475–487.

[14] G. Sang Ajang & F. Teer,
An Efficient Algorithm for Detection of Combined Occurrences,
Information Processing Letters, 8 (1979), p. 137–140.

[15] J. Schumann & R. Letz,
PARTHEO: A High-Performance Theorem Prover,
Proceedings of the 10th International Conference on Automated Deduction, 1990, pp. 40–56.

[16] J. Slaney,
SCOTT: A Model-Guided Theorem Prover,
Proceedings of the 13th International Conference on Artificial Intelligence, 1993, pp.109–114.

[17] J. Slaney & G. Meglicki,
MaGIC 2.0: Notes and Guide,
Technical report TR-ARP-1/91, Automated Reasoning Project, Australian National University, Canberra, 1991.

[18] J. Slaney, M. Stickel & M. Fujita,
Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems,
forthcoming.

[19] R. Sosič & J. Gu,
3,000,000 Queens in Under One Minute,
SIGART Bulletin 2 (1991), pp. 22–24.

[20] P. Thistlewaite, M. McRobbie & R. Meyer,
Automated Theorem Proving in Non-Classical Logics,
Pitman, London, 1988.

[21] L. Wos, R. Overbeek, E. Lusk & J. Boyle,
Automated Reasoning: Introduction and Applications,
Prentice-Hall, New Jersey, 1984.