Technical Report TR-ARP-12-94

Automated Reasoning Project
Research School of Information Sciences and Engineering
and Centre for Information Science Research
Australian National University

December 31, 1994

MINLOG

John Slaney

Abstract The program minlog is a theorem prover for propositional minimal and intuitionist logic, based on a cut-free sequent calculus. It achieves speed more through careful coding than by incorporating sophisticated logical techniques. It makes use of certain extra (derivable) rules to shorten proofs in trivial ways, and it avoids some searching by not working on the same sequent twice during the same proof search and by not trying to prove sequents which are not classical tautologies. Proofs may be displayed in a Fitch-style natural deduction format.

As well as being a tool for the logical investigation of constructive systems, minlog is put forward as representing the baseline for high performance theorem proving in this area.

The program, with sources, Makefile, documentation and 'man' page, is available by anonymous ftp from arp.anu.edu.au.

1 Overview

Minlog is a theorem prover for propositional minimal logic which can also prove theorems of Heyting's intuitionist logic. It implements a decision procedure based on a cut-free sequent calculus formulation of these systems. While the method it uses is rather unsophisticated, on small problems minlog is fast. It achieves speed by being carefully coded in a low-level language (C, in fact) and by eliminating many obvious redundancies in proof searches. In particular, it keeps a record of the irreducible sequents it has tried to prove during the current proof search (where a sequent is irreducible if the only rules by which it could have been obtained are non-invertible ones) so that it never tries to prove the same irreducible sequent twice in the one search. It also does not seek proofs of sequents which are not classically valid, and it checks for certain trivial features such as a pair of antecedents os the form A and $A \rightarrow B$ before searching any deeper for proofs of sequents.

The program is thus useful as a point of comparison. Since it is not an intelligent prover, but merely a "fast and dirty" one, it represents what can be done by brute force. The decision problem for the logics concerned is extremely intractable (P-space hard) so intelligence should triumph over brute force quite easily. Hence minlog should not be expected to deal as well with hard problems as other provers in its class insofar as these are worthwhile implementations of worthwhile ideas for proof search in constructive logics. Put contrapositively, any minimal logic or intuitionistic propositional prover that fails to beat minlog in at least some convincing sense is worthless.

2 Installation

Installing minlog should be extremely simple.

1. Obtain the sources either by anonymous ftp from arp.anu.edu.au or by writing to the author. Uncompress the file and then extract the contents by typing

   ```
   % tar -xvf minlog.tar
   ```

2. In directory minlog.1 edit the Makefile. Only the top few lines should need to be changed. Set BIN to the place where minlog's binaries are to live. If you intend to install the online manual page, set MANDIR and MANSECT accordingly.

3. To make the binary, type

   ```
   % make minlog
   ```

   To install the binary, type

   ```
   % make install_minlog
   ```

   To make the manual page, type

   ```
   make man
   ```

   To install the manual page, type

   ```
   make install_man
   ```

   Alternatively, a simple `make` makes both binary and manual, while `make install` installs them both, making them first if they don't already exist.

4. To clean up the source directory, type

   ```
   make clean
   ```

The documentation should be made with LaTeX. The source files for it are in directory minlog.1/doc and the top level file is called minlog.tex. If any of the above causes problems, please contact the author.

3 Logic

The intuitionist logic of Heyting occupies a special place among nonclassical logical systems. Not only is it one of the oldest to be fully articulated, it is also the one most deeply investigated and the one most elaborately and convincingly equipped with mathematical and philosophical underpinnings. The constructivist view of mathematics, and more ambitiously of all descriptive discourse, does not uniquely determine any one logic as the only acceptable theory of reasoning any more than does the classical realist outlook, but it strongly supports the departures from classical logic which lead to Heyting's calculus and it does suggest that calculus as an upper bound on what can be considered a reasonable logical system.[1] The present notes concern only a particular theorem prover for the propositional fragment of intuitionist logic and are not concerned to argue the case for or against constructivism either in mathematics or elsewhere in life. The reader seeking

[1] This is not to ignore the existence of constructive intermediate logics. The point is merely that the usual intuitionist system is the strongest to have a really smooth cut-free formulation, simple and intuitive semantics and the like.

that case may begin with the references given in the bibliography below [1, 4, 6] and bootstrap from there. The motivation for minlog was not so much to support a particular view of the formal sciences as to see what could be done with a particular style of automatic theorem proving and to do so with a logic in which there is a large amount of independent interest.

Almost as old as Heyting's full intuitionist calculus is the positive logic of Johansson of which Heyting's is a conservative extension most conveniently obtained by adding an absurd constant $\bot$ satisfying the axiom scheme

$$\bot \to A$$

and defining negation in the usual way

$$\neg A \ =\mathrm{df}\ A \to \bot$$

Minimal logic is the system which extends positive logic by adopting this definition of negation as implying The False but which avoids laying down any postulates which might determine which proposition The False should be. It may be axiomatised either in the natural way with a false constant or directly with axioms involving the unary negation operator.

These logics are perhaps the easiest of all to present in a cut-free sequent calculus version. By a *sequent* we mean a pair $\langle \Gamma, B \rangle$ where Γ is a set of formulae of propositional logic in the usual connectives and B is a single formula at most. Where Γ is finite, we shall write the sequent

$$A_1 \ldots A_n \ \vdash \ B$$

and read it at the meta-proposition that A_1 and $\ldots$ A_n together entail B. In the usual presentation, the *axioms* of the sequent calculus are the sequents

$$p \ \vdash \ p$$

for atomic p, or perhaps the analogous identity sequents for arbitrary formulae A. Minlog uses the more general axiom scheme

$$\Gamma, A \ \vdash \ A$$

This makes it unnecessary to specify or apply a separate rule of weakening. Since the collection on the left of the turnstile is a set, rather than a sequence, there is similarly no need for explicit structural rules of exchange or contraction. On the point of contraction, however, more will be said below.

The simplest connective to motivate and explain is conjunction. The compound $A \wedge B$ has exactly the inferential force of its two conjuncts together, so any set $\Gamma \cup \{A \wedge B\}$ entails exactly what is entailed by $\Gamma \cup \{A, B\}$,

and similarly Γ entails $A \wedge B$ iff it entails each of A and B. The sequent calculus rules for conjunction are therefore the expected ones:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \vdash \wedge \qquad\qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge \vdash$$

Disjunction is dual to conjunction, though this is slightly obscured by the need to formulate the rules with single formulae on the right:

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee \vdash \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vdash \vee \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

The remaining connective is implication, which is to be governed by the deduction theorem and its converse. This results in the rules:

$$\frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \to \vdash \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \vdash \to$$

The form of the rule $\to \vdash$ hides an important subtlety. It may happen that the implication $A \to B$ is a member of Γ. In the cases of all other rules this sort of thing does not matter, so the principal formula in each case may be assumed not to occur in Γ, but in this one case it does matter. There are sequents whose proof depends essentially on this possibility. Consider, for example the sequent

$$(p \vee (p \to q)) \to q \vdash q$$

Clearly this can only come by $\to \vdash$ since there is no other main connective. If we do not take Γ to be $A \to B$ in this case, we are left needing to prove

$$\vdash p \vee (p \to q)$$

and while this is a theorem of classical logic, it is not provable intuitionistically since it could only come by $\vdash \vee$ which is plainly impossible since neither p nor $p \to q$ is a theorem. The possibility that the implication on the left may have to be duplicated in order for $\to \vdash$ to apply is in a sense what makes the decision problem for these constructive logics harder than the analogous problem for classical logic. We shall return to it below.

Another subtle point is that since Γ occurs on the left of both premises of each two-premise rule, there may be more occurrences of a formula in the left sides of the (combined) premises than in the conclusion. Indeed, since Γ is a set, there can be only one in the conclusion. This facility

builds in a rule of contraction and is needed in proofs of such sequents as $p \to (p \to q) \ \vdash \ p \to q$ which is proved thus:

$$\dfrac{\dfrac{\boldsymbol{p} \ \vdash \ p \qquad\qquad \boldsymbol{p}, p \to q \ \vdash \ q}{p \to (p \to q), \boldsymbol{p} \ \vdash \ q}}{p \to (p \to q) \ \vdash \ p \to q} \qquad \dfrac{p \ \vdash \ p \qquad\qquad q \ \vdash \ q}{}$$

The bold occurrences of p illustrate the point, two of them in the premises being contracted to one in the conclusion.

So much for positive logic. To obtain minimal logic we simply add a constant $\bot$ to the language and regard negation as defined in the normal way. That is, $\neg A$ is simply an abbreviation for $A \to \bot$ so that the logic of negation in minimal logic is just part of that of positive implication. To obtain full intuitionist logic, we have to give $\bot$ the special property that it logically entails everything. This is easily done by adding more axioms

$$\Gamma, \bot \ \vdash \ A$$

As usual with sequent calculus presentations of logics like these, it is necessary to establish the admissibility of the rule 'Cut' in the form

$$\dfrac{\Gamma \ \vdash \ A \qquad\qquad \Delta, A \ \vdash \ B}{\Gamma, \Delta \ \vdash \ B} \ \text{cut}$$

This proof is routine. Note that since Γ and Δ are sets rather than sequences, repetitions of formulae cannot occur within them, so there is no need to go via a proof of Mix. In minlog's presentation of the logics, there are no explicit structural rules other than the axioms. In order to prove Cut, therefore, it is necessary to establish first that the structural rule of weakening holds in the form that if there is a cut-free proof of a sequent $\Gamma \ \vdash \ A$ then for any set Δ there is a cut-free proof *of the same length* of the sequent $\Gamma, \Delta \ \vdash \ A$. Proof of this lemma is by easy induction on the construction of the proof of $\Gamma \ \vdash \ A$. The admissibility of cut is then proved by the usual double induction on rank and degree and is case-ridden but easy. Details are left to the reader with time to kill.

Minlog performs a top-down proof search recursively seeking a proof of a given sequent $\Gamma \ \vdash \ A$. Some details are given in the next section. Briefly,

it looks at the main connective of A and those of the formulae in Γ and considers the rules which may have been applied last in a proof of the sequent. In each case the sequents which form appropriate premises for the rule are set up as subgoals and either proved or shown to be unprovable. If one of the subgoals (or sets of subgoals) succeeds, the sequent is proved; if they all fail, the sequent is shown to be unprovable. By inspection we may note that the calculus has the subformula property: all formulae which occur in the premises of any rule are subformulae of formulae in its conclusion. Evidently, therefore, every formula which occurs in the proof of $\Gamma \vdash A$ is a subformula of A or of some member of Γ, and equally evidently there are only finitely many such formulae and therefore only finitely many different sequents which can be built out of them. Hence the proof search tree branches finitely many ways at each point, and all non-looping branches in it are finite. Therefore, unless there are loops in branches, the proof search tree is finite and minlog's proof search is guaranteed to terminate in a finite time. Minlog, as might be expected, checks for loops in the current search branch and backtracks without reporting a proof if ever it finds itself considering a sequent which is already under test.

It is worth noting that the only possibility of generating a loop in the proof search is by applying $\rightarrow\vdash$ without removing the implication from the left side of the sequent, since all other rules have the property that the premises are strictly less complex than the conclusion.

In the interests of efficiency, minlog adds certain further rules, though these are all derivable anyway and hence do not change the set of provable sequents. First, and trivially, where $\rightarrow\vdash$ applies for free, it is used just to simplify the sequent:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A, A \rightarrow B \vdash C}$$

Next, and equally trivially, disjunctions are subsumed by their disjuncts and implications by their consequents:

$$\frac{\Gamma, A \vdash C}{\Gamma, A, A \vee B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, B, A \vee B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, B, A \rightarrow B \vdash C}$$

Slightly less trivially, minlog applies a rule which it calls *Double Negation*. This looks surprising at first, since of course double negation elimination is intuitionistically invalid in general. The rule is:

$$\frac{\Gamma, A \vdash B}{\Gamma, (A \rightarrow B) \rightarrow B \vdash B}$$

The name 'Double Negation', of course, comes from the fact that $\neg\neg A$ is by definition $(A \to \bot) \to \bot$ which is an instance of the $(A \to B) \to B$ of the rule. Note that the elimination is only warranted where the B concerned is the conclusion of the sequent. In practice this usually happens where B is $\bot$ anyway, so the name is a reasonable one.

There is one other justification which minlog sometimes uses for a line in a proof: that the corresponding sequent has already been proved during the present proof. Well, only an idiot proves the same theorem twice by the same method, so this is evidently fair enough.

4 Program

4.1 The algorithm

The implementation of the sequent calculus-based decision procedure in minlog is fairly straightforward. The principal function is a module proved which takes as argument a sequent and delivers as value a pointer to a proof of that sequent if there is one, or a null pointer if there is not. This function calls itself recursively in a fairly obvious manner to generate the subproofs.

The tests and attempted proofs of the sequent take place in the following order:

1. Check whether the sequent is an axiom or an absurdity like $\vdash p$. Return the trivial proof or failure if so.

2. If possible, apply one of the single-premise invertible rules $\vdash \to$, $\wedge \vdash$ or one of the extra rules noted in the last section. Return the proof or failure if so.

3. If possible, apply $\vdash \wedge$ or $\vee \vdash$ and return the proof or failure as appropriate.

4. Since there are no more invertible rules, check the sequent against the filters (see below). If it fails, return failure.

5. The sequent is irreducible. If it is already in the list of irreducible sequents currently under test, return failure. Otherwise add it to the list.

6. If the conclusion is a disjunction, try proof by $\vdash \vee$ and return the proof if successful. Do not return failure if unsuccessful.

7. For each remaining implicational formula on the left, try proof by $\rightarrow\vdash$ with the principal formula removed from the premises. Return the proof if one is found.

8. All else having failed, try each possible proof by $\rightarrow\vdash$ retaining the principal implicational formula among the premises. Return the proof if one is found.

9. Nothing has worked, so return failure.

On returning a proof or disproof of an irreducible sequent, of course, minlog deletes that sequent from the list of those under test. This is part of a very simple implementation of the looping check.

It remains to describe the filters. These are used to avoid trying to prove obviously invalid sequents and also to avoid proving the same thing twice. Each irreducible sequent which has been proved during the current proof search is stored as such, and each one which has been disproved is also recorded. Each "new" irreducible sequent is sought among the previously proved ones and among the previously disproved ones to see whether the labour of deciding it again can be avoided. The proved and disproved sequents are stored in two binary trees, so that the search for a match can be made in time logarithmic in the number of sequents stored. This recording of previous work appears to be very effective in reducing the proof search time in many cases. Minlog records only the irreducible sequents encountered during the current proof search; it does not carry these databases of sequents over from proof to proof.

The other filter is a test for provability in classical logic. It is much easier to test whether a sequent is valid in classical truth tables than whether it is provable in intuitionist or minimal logic. Minlog does not have a sophisticated classical propositional decision module, such as an implementation of the Davis-Putnam algorithm or the like, but performs a line-by-line truth table test. This is not as slow as it seems, for formulae with few variables. With fewer than 6 variables, there are at most 32 assignments of values, so the complete truth table for every subformula of the problem can be calculated just once, in advance of the search, in linear time and stored as a bit vector in one machine word. It can subsequently be looked up in constant time, which means essentially for free. With more than 5 variables, this hack would require a representation in more than one word per subformula, and minlog prefers to perform a simple-minded test instead, but clearly this module could be made more efficient if desired. It remains the case that truth table testing is only NP complete, while the decision problem for min-

imal logic is P-SPACE so problems which are hard by classical standards are unlikely to be encountered in feasible cases.

Clearly there is nothing magic about classical two-valued truth tables for this purpose. A stronger necessary condition for provability would be validity in a structure like the three-element Heyting lattice. However, the naïve test in this structure takes $\mathcal{O}(3^n)$ time, where there are n variables, instead of the $\mathcal{O}(2^n)$ needed for classical logic. This difference makes two-valued testing faster though dirtier, and minlog prefers speed on small problems to all else.

Since the filters only remove unprovable sequents (in fact, they hit most of them, according to the experimental trials to date) their use in no way affects the proofs of provable sequents. It simply makes discovery of those proofs feasible.

4.2 Format of Input

A sequent is presented to minlog as a string consisting of the antecedents, separated by commas if there is more than one, then a colon (optional if there are no antecedents), then the consequent. Formulae are written in infix notation. Parentheses around the whole formula are not needed, and to reduce the number of necessary parentheses within formulae, note that the connectives are ranked

$$\neg \quad \wedge \quad \vee \quad \rightarrow$$

in increasing order of scope (decreasing tightness of binding). Finally, a period within a formula represents a left parenthesis whose right mate is to be imagined as far to the right as is reasonable. Negation is written as tilde, conjunction as ampersand, disjunction as upper case 'V' and implication as the two-character sequence 'minus' followed by 'greater-than'. The 'minus' may be omitted. Each sequent is terminated by a line feed (RETURN). End of input is signalled by input of the null sequent which is simply a line feed. Any lower case letter may be used as a propositional variable.

It is worth noting that any subformula of the form $\neg A$ is converted by the parser into $A \rightarrow F$ since that is the definition of negation. The rules applied to negations are therefore the implication rules. For output, the deparser converts any subformula of the form $A \rightarrow F$ into the corresponding negation $\neg A$.

4.3 Format of Output

A proof in the sense of §3 is an abstract object: a tree whose nodes are sequents (ordered pairs each consisting of a set of formulae and a formula) and whose arcs inferences. How one chooses to write a physical representation of the proof is not intrinsic to it. Minlog writes it as a natural deduction proof after the manner of Fitch. The correct view of this matter is not that it can transform the Gentzen proof into a Fitch proof but that the Fitch proof *is* the Gentzen proof in a particular format.

Consider, for example, this proof of the theorem $\neg\neg(p \vee \neg p)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}}{p \vdash p \vee \neg p} \qquad \dfrac{}{\bot, p \vdash \bot}}{\neg(p \vee \neg p), p \vdash \bot}}{\neg(p \vee \neg p) \vdash \neg p}}{\neg(p \vee \neg p) \vdash p \vee \neg p} \qquad \dfrac{}{\bot \vdash \bot}$$

$$\dfrac{\neg(p \vee \neg p) \vdash \bot}{\vdash \neg\neg(p \vee \neg p)}$$

It may be written just by listing the sequents, thus:

(1)	$p \vdash p$	axiom
(2)	$p, \bot \vdash \bot$	axiom
(3)	$p \vdash p \vee \neg p$	$1 \vdash \vee$
(4)	$\neg(p \vee \neg p), p \vdash \bot$	$2, 3 \rightarrow \vdash$
(5)	$\neg(p \vee \neg p) \vdash \neg p$	$4 \vdash \rightarrow$
(6)	$\neg(p \vee \neg p) \vdash p \vee \neg p$	$5 \vdash \vee$
(7)	$\bot \vdash \bot$	axiom
(8)	$\neg(p \vee \neg p) \vdash \bot$	$6, 7 \rightarrow \vdash$
(9)	$\vdash \neg\neg(p \vee \neg p)$	$7 \vdash \rightarrow$

An alternative notation which is generally easier to read uses elimination rules instead. In this notation, only one formula per line is written. The antecedents of the sequent to be proved are written first as assumptions. Then each subproof which begins with the addition of a formula to the

antecedents is written as an indented column with the new assumption at the top and the succedent of the sequent at the bottom. For clarity, some rules such as Double Negation and $\wedge\vdash$are recorded without indentation of the subproofs. It is worth noting how the rule $\rightarrow\vdash$is treated. Suppose we are writing out the proof of a sequent in which $A\rightarrow B$ occurs on the left and is introduced at some point by an application of $\rightarrow\vdash$. At that point, in the Fitch-style listing of the proof, $A\rightarrow B$ will occur somewhere already, either as an assumption or as a result of something like a $\wedge\vdash$step. What we may do, therefore, is list the derivation of A from the other antecedents, then add a line deriving B from the $A\rightarrow B$ and the A by *modus ponens* and then append the derivation of the succedent of our sequent from the antecedents plus$B$. Here is an example:

(1)	$\lvert\ \neg(p \vee \neg p)$	ASSUMPTION
(2)	$\lvert\ \lvert\ p$	ASSUMPTION
(3)	$\lvert\ \lvert\ p \vee \neg p$	$2, \vdash \vee$
(4)	$\lvert\ \lvert\ \bot$	$1, 3, \rightarrow\vdash$
(5)	$\lvert\ \neg p$	$2, 4, \vdash\rightarrow$
(6)	$\lvert\ p \vee \neg p$	$5, \vdash \vee$
(7)	$\lvert\ \bot$	$1, 6, \rightarrow\vdash$
(8)	$\neg\neg(p \vee \neg p)$	$1, 7, \vdash\rightarrow$

Note that in the present release, minlog is a little rough in presenting proofs like this. It omits the annotation recording which previous lines serve as iputs to each rule application, though it does name the rules used.

4.4 Verbosity and other settings

Minlog gives a choice of four verbosity settings, from 0 to 3.

−v 0 None. This setting is self-explanatory.

−v 1 At the end of the execution a profile is printed showing the number of times each rule has been applied. The times taken to decide the most difficult sequents are also listed.

−v 2 The profile is additionally printed for each individual sequent.

−v 3 Maximal verbosity. In addition to the above, every time the function proved is entered the sequent is printed, and every exit from proved is recorded with the reason. This enables the entire proof search to be traced.

Besides the verbosity settings, there are three other command line options.

−p By default, minlog is oracular, reporting only that the sequent is provable or that it is not provable. The switch `-p` causes a proof to be printed in the natural deduction format described above.

−i Also by default, as its name suggests, the program searches for proofs in minimal logic. The command line option `-i` turns on the rule $\bot\vdash$, causing the search to be for proofs in intuitionist logic.

−g As a pretty extra, minlog can be made to produce a file `bargraph.tex` which contains the LaTeX source code for some histograms of the summary profile. The first of these shows the percentace of calls to proved which exit for each reason (the percentage of sequents which are axioms, the percentage which are proved by $\vdash\rightarrow$ and so forth). Unfortunately, in the present release, the bars of the histogram are not labelled, so you have to refer to the printed profile to interpret the graph. The second histogram shows the same numbers, but as percentages of the number of times the search gets as far as potentially applying each rule or filter. So for example, the second bar (the number of successful applications of the rule $\bot\vdash$) is given as a percentage not of the number of calls to proved but of that number minus the number which were minimal logic axioms. The third histogram is another picture of the profile, this time showing the number of sequents which get as far as the potential application of each rule (that is, the number of logged events remaining at each stage). The `-g` option turns on the graphing, independently of the verbosity level.

5 Performance

No very systematic evaluation of minlog's performance has been made, but a large problem set was used to help fine-tune the program. This consisted of 20,000 three-variable problems mostly with about 100–150 subformula occurrences in each problem. None of these problems is especially difficult for minlog, perhaps because of the small number of variables involved, but the problem set is worth noting because large-scale testing of theorem provers against such sets is still quite rare.

The problems in this set were all sequents of the forms

$$p\star(q\star r)\ \vdash\ (p\star q)\star r$$

```
Initial:   F_1 ← F
           F_2 ← p
           F_3 ← q
           next ← 4

Loop:      for each i < next do
               for each j ≤ i do
                   Try(F_i ∧ F_j)
                   Try(F_i ∨ F_j)
                   Try(F_i → F_j)
                   Try(F_j → F_i)

Procedure Try(A: formula)
           if A is new and C(A) is new then
               F_next ← A
               next ← next + 1
```

Figure 1: Generating binary operations

and

$$(p \star q) \star r \ \vdash \ p \star (q \star r)$$

where $\star$ is a formula in two variables definable in the propositional language. For example, if $A \star B$ is chosen to be $\neg(A \to B)$ then the first of the two sequents is

$$\neg(p \to \neg(q \to r)) \ \vdash \ \neg(\neg(p \to q) \to r)$$

and of course the second is just its converse. The $\star$ operations used were all non-trivially different and were generated as shown in Figure 1.

A formula counts as "new" if it is not equivalent (in minimal logic) to any formula F_k for $k <$ next. Formula $\mathcal{C}(A)$ is just A with p substituted for q and q substituted for p throughout. It is obvious that the set of associative functions is closed under $\mathcal{C}$ so it was felt that operations equivalent to the results of commuting each other were not sufficiently "new" for the purposes at hand.

In order to make the check for newness feasible, generated formulae were divided into equivalence classes according to their three-valued truth tables in the three-element Heyting lattice. It was then necessary only to check for equivalences within each class. Minlog itself was used to perform the remaining checks. It was quick, since the formulae defining the binary

operations are much shorter that the associativity claims to which they give rise.

When ten thousand different operations had been generated, the generation process was stopped and the tests for associativity made. Complete results were not kept—the idea was not to compile statistics but merely to fine-tune the prover for speed—but a "hall of fame" of the hardest provable cases and the hardest disprovable ones was compiled and repeatedly used.

References

[1] M. Dummett, **Elements of Intuitionism**, Oxford, OUP, 1977.

[2] R. Dyckhoff, *Contraction-free Sequent Calculi for Intuitionistic Logic*, **Journal of Symbolic Logic** 57 (1992) pp. 795–807.

[3] F. Fitch, **Symbolic Logic**, New York, Ronald Press, 1952.

[4] A. Heyting, **Intuitionism, an Introduction**, Amsterdam, North-Holland, 1956.

[5] P. Thistlewaite, M. McRobbie and R. Meyer, **Automated Theorem-proving in Non-classical Logics**, London, Pitman, 1984.

[6] van Dalen *Intuitionist logic*, D. Gabbay & F. Günthner (eds), **Handbook of Philosophical Logic** Vol. 3, Dordrecht, Reidel, 1986.