SICS Research Report
R87010B

ISRN : SICS-R—87/10B--SE
ISSN : 0283-3638

Algorithmic Aspects of Intuitionistic Propositional Logic

by

Torkel Franzén

1988

Swedish Institute of Computer Science
Box 1263, S-164 29 KISTA, SWEDEN

Algorithmic aspects of intuitionistic propositional logic

by
Torkel Franzén

SICS R87010B
Research Report
ISSN 0283-3638

Algorithmic aspects of intuitionistic propositional logic

Torkel Franzén
Swedish Institute of Computer Science
Box 1263 S-164 28 Kista SWEDEN

1 Introduction

Intuitionistic propositional logic is well known to be decidable, but there seems to be a dearth of practical algorithms in the logical literature. (See [1] for an overview of intuitionistic logic and its literature.) Hence the present report, dealing with some ways of making the standard decision procedure for the sequent calculus (presented e.g. in [2]) more efficient.

The reader will naturally ask what applications of intuitionistic propositional logic motivate a preoccupation with efficient decision methods. To this I can only answer that I don't know of any such applications. My own preoccupation with intuitionistic logic stems from a project at SICS in which the possibility of using the intuitionistic sequent calculus in logic programming is investigated. Of course, anybody who is interested in theorem proving in full intuitionistic predicate logic will have occasion to consider propositional algorithms. For background to the present report, see [3].

2 Syntax and rules

The formulas of intuitionistic propositional logic are constructed from atomic formulas p,q,r,.. and - in the formulation adopted here - the special atomic formula $\bot$, using parentheses and the connectives $\wedge,\vee,\supset$. $\bot$ (falsum or the absurdity) is interpreted as a logically false statement, and the negation $\neg A$ of a formula A is accordingly defined as an abbreviation of $A\supset\bot$. Equivalence is also introduced as an abbreviation: $A\equiv B$ stands for the formula $(A\supset B)\&(B\supset A)$. The letters A,B,C,D will be used to denote formulas; Γ and Δ will denote finite sets or sequences of formulas. Parentheses will be omitted in accordance with the following conventions: 1) the outermost parentheses are omitted when formulas occur in isolation; 2) association to the right is used for $\vee$ and &; 3) $\vee$ and & bind harder than $\supset$ and $\equiv$. Thus e.g. $A\vee B\supset C\&D\&E$ stands for $((A\vee B)\supset(C\&(D\&E)))$, whereas $A\&B\vee C$ and $A\supset B\supset C$ are undefined.

The semantical relation $\Gamma \models A$ (A is a logical consequence of the formulas in Γ) was first formalized by A.Heyting through an axiomatization of intuitionistic (predicate) logic based on an informal interpretation of the connectives ("the Heyting interpretation"). To give a formal completeness proof for the resulting axiomatization relative to the Heyting interpretation is difficult for various reasons (see [2]). Intuitionistic predicate logic has however been proved complete with respect to formal set-theoretical (not intuitionistically valid) interpretations such as that given by Kripke models. In the exposition below, no formal semantics will be used and all proofs will be wholly syntactical. The Heyting interpretation will be used informally at some points, but it is open to the reader to take "A is a logical consequence of Γ" and "$\Gamma \rightarrow A$ is logically valid" as alternative ways of saying that $\Gamma \rightarrow A$ is provable in the sequent calculus.

The intuitionistic consequence relation $\Gamma \models A$ is not decidable by means of truth value calculations. This follows by a simple argument due to Gödel. If intuitionistic validity can be reduced to truth tables there is a set of truth values $\{1,..,n\}$, a subset D of designated truth values (corresponding to Truth in the classical case) and truth tables for the connectives such that a formula A is intuitionistically valid if and only if it is a tautology, i.e. its truth value V(A) under V belongs to D for every valuation V. The following formula A (using an obvious abbreviation) must then be a tautology:

$$\bigvee_{1 \leq i < j \leq n+1} p_i \equiv p_j$$

To see that A is a tautology, note that for any V we must have $V(p_i)=V(p_j)$ for some $i<j$ since there are more propositional variables than truth values. Say $V(p_1)=V(p_2)$. For some function F we have $V(A) = F(V(p_1),V(p_2),..V(p_{n+1})) = F(V(p_1),V(p_1),..V(p_{n+1}))$ $= V(A(p_1/p_2)) \in D$, since the formula $A(p_1/p_2)$ obtained by substituting p_1 for p_2 in A is intuitionistically valid. Thus A is a tautology. A is not intuitionistically valid, however. This follows from the fact that intuitionistic logic has the constructive disjunction property, i.e. if $A \vee B$ is intuitionistically valid, then A or B is intuitionistically valid.

That intuitionistic propositional logic is nevertheless decidable follows easily from the formalization given in (the cut-free version of) Gentzen's sequent calculus, in which one derives expressions of the form $\Gamma \rightarrow A$ called sequents. Γ is the antecedent of the sequent, A the consequent. That the sequent $\Gamma \rightarrow A$ is *valid* means that A is a logical consequence of the formulas in Γ. When the letters Γ and Δ are used in sequents they will be assumed to denote finite sequences (lists) of formulas (including the empty sequence) rather than finite sets. Γ,Δ stands for the concatenation of Γ with Δ, and similarly for A,Γ, etc. Sequents $\Gamma \rightarrow A$ and $\Gamma' \rightarrow A$ are *set-equivalent* if Γ and Γ' are equal regarded as sets.

An axiom in the calculus has the form

$$\bot,\Gamma\rightarrow C \quad (\bot\text{-axiom})$$

or

$$B,\Gamma\rightarrow B \quad \text{(logical axiom)}$$

The rules of the calculus are the following:

$$\frac{A,B,\Gamma\rightarrow C}{A\&B,\Gamma\rightarrow C} \qquad \frac{\Gamma\rightarrow A \quad \Gamma\rightarrow B}{\Gamma\rightarrow A\&B}$$

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

$$\frac{A\supset B,\Gamma\rightarrow A \quad B,\Gamma\rightarrow C}{A\supset B,\Gamma\rightarrow C} \qquad \frac{A,\Gamma\rightarrow B}{\Gamma\rightarrow A\supset B}$$

The intuitionistic system defined by these axioms and rules will be referred to as G in the following. It differs from standard formulations mainly in not including the contraction rule:

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

Instead the unavoidable use of contraction has been incorporated into the formulation of the $\supset\rightarrow$-rule. A Kripke semantics completeness proof for a system incorporating G (from which its equivalence to standard formulations follows) is given in [3].

Formal proofs in the sequent calculus are trees rooted in the conclusion, with axioms as tips and every other node related to its immediate successor(s) by one of the inference

rules. By a proof of a formula A will be meant a proof of the sequent $\rightarrow$A. There are two logical rules for each of the connectives: one for introducing the connective on the left side of a sequent, i.e. in the antecedent, and one for introducing it on the right side, in the consequent. The rules are accordingly named $\&\rightarrow$, $\rightarrow\&$ etc. The $\rightarrow\vee$-rule, it will be noted, has two forms.

To reduce verbosity, the axioms and rules are formulated above with the exhibited formulas leftmost in the antecedent. The rules are to be understood, however, as covering every permutation of the formulas in the antecedent. For example, $\&\rightarrow$ covers every step of the form

$$\frac{\Gamma\rightarrow C}{\Gamma'\rightarrow C}$$

where Γ is a permutation of A,B,Δ and Γ' is a permutation of A&B,Δ. Similarly in arguments about sequents. Another way of putting this is that Γ and Δ stand for "multi-sets", i.e. structures of the type obtained by identifying a list with its permutations. This latter interpretation is not always appropriate, however. Consider the proof

$$\frac{\dfrac{p,q,r,p,q,r\rightarrow r \qquad p,q,r,p,q,r\rightarrow q}{p,q,r,p,q,r\rightarrow r\&q}}{p\&q,r,p,q,r\rightarrow r\&q}$$

If we regard the sequents as multisets, the conjunction p&q in the conclusion cannot be associated with any particular occurrences of p and q in the premiss, and similarly for r and q in the axioms. Generally speaking, we cannot follow an occurrence of a formula in the conclusion upwards through the proof. To introduce such an association of formula occurrences is to impose extra structure on the proof, structure which can contain valuable information (see §10). Accordingly, when this extra structure is appealed to we must assume that an application of a rule is accompanied by an indication of which formula occurrences in the premisses and conclusion enter into the application. This will be referred to as an *occurrence analysis* of the proof.

Minimal logic is obtained by leaving out the $\perp$-axioms. *Contraction-free* (intuitionistic or minimal) logic is obtained by changing the left premiss of the $\supset\rightarrow$-rule from A$\supset$B,$\Gamma\rightarrow$A to $\Gamma\rightarrow$A. Minimal and intuitionistic logic are equivalent as far as

computational complexity is concerned, and the difference between the two plays no role in any of the algorithms presented here. Contraction-free logic, on the other hand, is essentially simpler than minimal or intuitionistic logic.

3 The decidability of intuitionistic propositional logic

All systematic procedures for finding a proof of a given sequent in the cut-free sequent calculus (whether in propositional logic or in full predicate logic) proceed by attempting to construct a proof-tree in a bottom-up fashion starting from the conclusion. In fact when I speak in the following of applying one of the rules, a "backwards" application is always intended: a step from the conclusion to the premiss or premisses. In such a step, the formula to which the rule is applied is said to be *used*. Disregarding all questions of efficiency, we see by inspecting the rules that a sufficiently systematic procedure of this kind will decide whether or not a given sequent is provable in the system G. We start with a sequent S and the empty path <>. To prove a sequent S with path τ, establish that S is an axiom or apply an inference rule to S to obtain one or two premisses with path $\tau \cup \{S\}$, at least one of which is not set-equivalent to any sequent in $\tau \cup \{S\}$. Then prove the premisses. If S is not an axiom and no such premisses are obtainable, backtrack to the most recent sequent in τ where a new choice of formula to use can be made. That this procedure terminates is guaranteed by the fact that every formula occurring in the premisses of an application of a rule is a subformula of one of the formulas in the conclusion.

To make the procedure less unfeasible we want to reduce the backtracking and avoid having to check whether a sequent is set-equivalent to some sequent in a path. The second part of this is the subject of §4. The first (and standard) step towards reducing the backtracking follows from the observation that we need never consider any alternative to an application of an *invertible* rule, i.e. one with the property that the conclusion is provable if and only if the premisses are provable. The rules $\rightarrow\&$, $\rightarrow\supset$, $\vee\rightarrow$, and $\&\rightarrow$ are invertible, and the $\supset\rightarrow$-rule is semi-invertible: if the conclusion is provable, then the right premiss (but not in general the left) is provable. For completeness, this semantically obvious fact is stated here as the:

Invertibility lemma. The rules $\rightarrow\&$, $\rightarrow\supset$, $\vee\rightarrow$, $\&\rightarrow$ are invertible, and the $\supset\rightarrow$-rule is invertible w.r.t. the right premiss.

Proof: The lemma follows by an induction on proofs. ■

An application of an invertible rule will be called a *reduction* and a sequent $\Gamma\rightarrow A$ to

which no reduction applies - i.e. A is either atomic or a disjunction and all formulas in Γ are atomic or implications - will be called a *reduced* sequent.

4 Contraction

The possibility of applying the inference rules ad infinitum and the resulting complication in the decision procedure is due to the $\supset\rightarrow$-rule alone, since this rule incorporates contraction, i.e. the formula $A\supset B$ occurs both in the conclusion and in the left premiss sequent. In the bottom-up construction of proofs this means that an implication on the left side of a sequent can be used any number of times (whereas conjunctions and disjunctions are dissolved and vanish). Contraction need not be incorporated in invertible rules and may or may not be needed in non-invertible rules[1]. The need for contraction in the $\supset\rightarrow$-rule is demonstrated by the formula

(*) $$\neg\neg(\neg p_1 \vee ... \vee \neg p_n \vee (p_1 \& ... \& p_n))$$

In a proof of (*), the formula obtained by deleting the first negation sign must be used n+1 times and thus n contractions are needed. These formulas are in fact valid in minimal logic, so (*) can be replaced by the less immediately intelligible

$$((p_1 \supset q) \vee ... \vee (p_n \supset q) \vee (p_1 \& ... \& p_n) \supset q) \supset q$$

Instead of checking that a further application of $\supset\rightarrow$ will yield a new sequent (as in the algorithm of §3), we can make use of an upper bound on the number of times we need to use an implication in a proof. That there is such a computable upper bound follows from the decidability of the calculus.[2] It is possible to formulate an easily computable upper bound - the function bc given below - which allows us to dispose quickly of implications in many simple cases. A general-purpose algorithm, however, works faster by checking that a contraction is potentially useful, and this is the method to be adopted here. First some definitions.

When derivations and branches in derivations are considered in the following, it will always be from the bottom of the derivation upwards. Thus branches are said to turn left or right at an application of a two-premiss rule, and of two sequents in a branch the one further from the root is said to occur later in the branch than the one nearer the root. A *transfer* in a branch is an application of $\rightarrow\supset$; the antecedent A of the implication used in

[1] Here I have in mind the rules of G with $\supset\rightarrow$ amended so as not to incorporate contraction.

[2] In the case of full predicate logic, the undecidability of intuitionistic logic without the universal quantifier (in which classical predicate logic is interpretable) implies that there is no computable upper bound on the number of implication contractions needed to prove a formula.

the application is the *transferred* formula. A *switch* in a branch is a left turn at an application of $\supset\rightarrow$. Γ *covers* the formula A if A occurs in Γ, or A is $B\vee C$ and Γ covers B or C, or A is $B\&C$ and Γ covers B and C, or A is $B\supset C$ and Γ covers C. Γ covers Δ if Γ covers every formula in Δ. Note that if Γ covers A then $\Gamma \models A$, and (by inspection of the rules) if $\Gamma'\rightarrow A'$ occurs after $\Gamma\rightarrow A$ in some branch of a proof, then Γ' covers Γ.

The set comp(A) of *components* of a formula A is defined by

$$\begin{aligned}
&\mathrm{comp}(A\vee B) = \mathrm{comp}(A\&B) = \mathrm{comp}(A) \cup \mathrm{comp}(B),\\
&\mathrm{comp}(A\supset B) = \{A\} \cup \mathrm{comp}(B),\\
&\mathrm{comp}(A) = \varnothing \text{ for atomic A.}
\end{aligned}$$

If A is $B_1\supset(B_2\supset\ldots\supset C)$ where C is not an implication, the formulas $B_1,\ldots B_n$ are said to form a *block* of components of A. Thus the set bcomp(A) of component blocks of A is defined by

$$\begin{aligned}
&\mathrm{bcomp}(A\vee B) = \mathrm{bcomp}(A\&B) = \mathrm{bcomp}(A) \cup \mathrm{bcomp}(B),\\
&\mathrm{bcomp}(B_1\supset(B_2\supset\ldots\supset C)) = \{\{B_1,\ldots B_n\}\} \cup \mathrm{bcomp}(C) \text{ if C is not an implication,}\\
&\mathrm{bcomp}(A) = \varnothing \text{ for atomic A.}
\end{aligned}$$

Now suppose $A\supset B$ is used at least twice in some branch:

$$\dfrac{\dfrac{A\supset B,\Gamma'\rightarrow A \quad B,\Gamma'\rightarrow C'}{(2)\quad A\supset B,\Gamma'\rightarrow C'}}{\vdots}$$

$$\dfrac{(1)\quad A\supset B,\Gamma\rightarrow A \quad B,\Gamma\rightarrow C}{A\supset B,\Gamma\rightarrow C}$$

It is clear - but requires a proof - that going in this way from $A\supset B,\Gamma\rightarrow A$ to $A\supset B,\Gamma'\rightarrow A$ is necessary only if Γ' contains new information obtained from components of A transferred between (1) and (2). Accordingly we define $A\supset B$ to be *over-contracted* in a branch if every component of A (if any) transferred between (1) and (2) in a segment of the branch of the form indicated above is covered by Γ. We thus need the following

Contraction lemma. If $\Gamma\to D$ is provable, there is a proof of $\Gamma\to D$ in which no implication is over-contracted.

For the proof of the contraction lemma, we need an operation described in the proof of the following (semantically obvious)

Covering lemma. Suppose $A,\Gamma\to B$ is provable and Γ covers A. Then $\Gamma\to B$ is provable.

Proof: By induction on the provability of $A,\Gamma\to B$.

First suppose $A,\Gamma\to B$ is an axiom. If it is a $\bot$-axiom then so is $\Gamma\to B$. Otherwise B occurs in A,Γ. If B occurs in Γ, $\Gamma\to B$ is again an axiom. If B is A we must show that $\Gamma\to A$ is provable whenever Γ covers A. Here we use induction on the covering relation. If A occurs in Γ then $\Gamma\to A$ is an axiom. If A is $A_1\&A_2$ and Γ covers A_1 and A_2, $\Gamma\to A_1$ and $\Gamma\to A_2$ are provable by the induction hypothesis, so $\Gamma\to A$ is provable using $\to\&$. Similarly for the other two cases. For the implication case we must note that it holds trivially in G that $E,\Gamma\to F$ is provable whenever $\Gamma\to F$ is provable.

Now suppose B is $B_1\&B_2$ and $A,\Gamma\to B$ follows by $\to\&$ from $A,\Gamma\to B_1$ and $A,\Gamma\to B_2$. By the induction hypothesis, $\Gamma\to B_1$ and $\Gamma\to B_2$ are provable, so $\Gamma\to B_1\&B_2$ follows by $\to\&$. The cases $\to\vee$ and $\to\supset$ are equally straightforward.

Now take the case when $A,\Gamma\to B$ follows by $\supset\to$. If A is not the formula used in this step, the induction hypothesis yields directly that $\Gamma\to B$ is provable. So suppose A is $E\supset F$ and $A,\Gamma\to B$ follows from $A,\Gamma\to E$ and $F,\Gamma\to B$. Γ covers $E\supset F$, so there are two cases. If Γ is $E\supset F,\Gamma'$ then F,Γ' covers F,Γ and by the induction hypothesis $\Gamma\to B$ follows by $\supset\to$ from $\Gamma\to E$ and $F,\Gamma'\to B$. If Γ covers F, the induction hypothesis applied to $F,\Gamma\to B$ yields that $\Gamma\to B$ is provable. The cases $\&\to$ and $\vee\to$ are similar. ■

Proof of the contraction lemma. Pick a proof of $\Gamma\to D$, and consider the steps of the form

$$\frac{E,\Delta\to F}{\Delta\to E\supset F}$$

where Δ covers E - these will be called *covered transfers*. We want to transform the proof so that the formulas transferred in covered transfers are never used. For this we apply (if E is used) the operation in the proof of the covering lemma to obtain a proof of $\Delta\rightarrow F$; then add E so as to obtain a proof of $E,\Delta\rightarrow F$ in which the exhibited formula E is not used. Here we must note that the only new covered transfers introduced by that operation (viz. in the case when $A,\Gamma\rightarrow B$ is an axiom) have the property that the transferred formula is not used.

This achieved, consider two consecutive applications of $\supset\rightarrow$ to $A\supset B$:

$$\dfrac{\dfrac{A\supset B,\Gamma'\rightarrow A \quad B,\Gamma'\rightarrow C'}{\vdots}\ (2)}{\vdots}\ (1)$$

$$\dfrac{A\supset B,\Gamma\rightarrow A \quad B,\Gamma\rightarrow C}{A\supset B,\Gamma\rightarrow C}$$

where every component of A transferred between (1) and (2) is covered by Γ, and hence (after the operation above) is never used in the proof. We get rid of one of the two application of $\supset\rightarrow$ to $A\supset B$ by one of the following two transformations. First, in case there is no switch between (1) and (2), replace the derivation above by

$$\dfrac{\begin{array}{c}A\supset B,\Gamma'\rightarrow A\\ \vdots\end{array} \quad A\supset B,\Gamma\rightarrow A \quad B,\Gamma\rightarrow C}{A\supset B,\Gamma\rightarrow C}$$

obtained by simply leaving out every application of $\rightarrow\vee$, $\rightarrow\&$, $\rightarrow\supset$ between (1) and (2). The result is still a proof since no transferred formula is used. In the case $\rightarrow\&$, this entails throwing away the the premiss derivation not leading to $A\supset B,\Gamma'\rightarrow A$. In case there is a switch between (1) and (2), replace the derivation by

$$\frac{A\supset B,\Gamma'\to A \quad B,\Gamma'\to C'}{A\supset B,\Gamma'\to C'}$$

$$\vdots$$

$$A\supset B,\Gamma\to C$$

obtained by leaving out every application of $\to\vee$, $\to\&$, $\to\supset$ before the first such switch. Note that these operations do not introduce any new applications of inference rules. Hence they lead to a proof without over-contracted implications. ■

The contraction lemma establishes that we need not use an implication $A\supset B$ in the antecedent of a sequent twice without an intervening transfer of a new component of A. Thus the number of components of A yields an upper bound on how many times we need to contract $A\supset B$ (in any one branch in a proof). In fact, since it is easily shown that we can always transfer all the formulas in a block of components at the same time, the number of blocks of components of A yields such an upper bound. Thus one algorithm consists in tagging implications with a counter which is decreased every time the implication is used, avoiding all comparisons of formulas or sequents. In cases requiring no or few contractions this is efficient, but as bc(A) increases, the algorithm of §5 will be very much faster. This algorithm does not count the contractions, but ensures that an implication is not used again until a new component has been transferred.

The number[3] bc(A) of blocks of components of a formula A is given by

$bc(A\supset(B\supset C)) = bc(B\supset C)$
$bc(A\supset B) = 1+bc(B)$ if B is not an implication
$bc(A\vee B) = bc(A)+bc(B)$
$bc(A\&B) = bc(A)+bc(B)$
$bc(A) = 0$ for atomic A.

Thus we need to use an implication $A\supset B$ at most bc(A)+1 times. There seems to be no essential improvement on bc as a simple syntactical upper bound on the number of contractions. In particular, two early conjectures turned out to be incorrect. First, the idea that we can use bc(A&B)=max(bc(A),bc(B)) is incorrect since the formula

[3] Here different occurrences of formulas are counted as different components. We get a possibly smaller measure bc*(A) by counting formulas rather than occurrences of formulas, at the cost of complicating the computation.

$$\neg\neg((p_1 \vee \neg p_1)\&..\&(p_n \vee \neg p_n))$$

requires n contractions. Also we cannot replace 1+bc(B) by max(1,bc(B)), since the sequent

$$(((p \supset s) \supset k \vee (p\&q) \vee (q \supset p\&s)) \supset k \vee (p\&q) \vee (q \supset p\&s)) \supset s \rightarrow s$$

requires two contractions.

A natural idea is to avoid the use of contraction altogether by introducing auxiliary variables and thus simplifying the antecedents of implications. For example, we see that the formula $\neg\neg(\neg p \vee \neg q \vee (p\&q))$ is valid if and only if the sequent

$$r \equiv \neg p, s \equiv q \rightarrow \neg\neg(r \vee s \vee (p\&q))$$

is valid, and this latter sequent is provable without contraction. It is interesting to note that the use of contraction cannot be completely eliminated in this way, for the sequent

$$a\&p \supset q \vee k,\ (p \supset q) \supset k,\ (a \supset k) \supset q \rightarrow q$$

is not provable without contraction and cannot be simplified by the introduction of auxiliary variables. However, this is no loss, since (as will be argued below) contraction is much less of a computational problem than the need to choose the proper implication in applications of the $\supset\rightarrow$-rule. Thus it is a poor bargain to exchange contractions for a whole bunch of new implications.

5 A basic algorithm

From the contraction lemma and the invertibility lemma the following decision procedure emerges. To prove a sequent S, establish that S is an axiom or apply any invertible rule and prove the premiss(es). If no invertible rule is applicable, apply a non-invertible rule and prove the premiss(es). An attempt to prove a sequent which is not an axiom fails when no rule applies: then we backtrack to the most recent point, if any, where a new choice of formula to which to apply a non-invertible rule can be made.

The following special directives apply: In the case of $\supset\rightarrow$, prove the right premiss first; if this fails, reject the conclusion as unprovable. When proving the left premiss, put the implication $A \supset B$ on ice - i.e. do not use it again - until a component of A has been transferred which is not covered by the antecedent. If a switch occurs before a new

component of A has been transferred, the implication is thrown away. A⊃B is referred to as the *provisional* implication; at each step in the procedure there is at most one provisional implication among the premisses of the sequent to be proved.

An obvious economy completes the specification of the procedure. In an application of →⊃ to Γ→A⊃B we must check whether A is covered by Γ, at least when there is a provisional implication. Just throw away A if A is covered by Γ and continue with Γ→B.

The condition for re-using A⊃B stipulated above yields what will be called the covering version of the algorithm. An alternative and more liberal condition allows us to use A⊃B again after having transferred a component of A which has not previously been transferred (from A or in any other context). This yields the transfer-list version of the algorithm, in which a list is kept of transferred formulas as the proof is constructed and the condition of not being covered by the formulas in the antecedent is replaced by the condition of not belonging to the list. The transfer-list version entails less checking, requires more space, and accepts more transfers than the covering version. In the Prolog implementations and tests in the Appendix the first factor seems to be dominant, since the transfer-list version was faster.

The procedure described above will be referred to as "the basic algorithm". The order in which to apply the rules is not specified in this description, apart from the stipulation that invertible rules are tried first. Nothing is said concerning the order in which to use the formulas in the antecedent. Nor is it specified whether the premisses are to be proved in parallel, in some fixed order, or according to some other scheme, except for the stipulation that the right premiss in an application of ⊃→ is proved first. In the Prolog implementation of the procedure given as icalc1 in the Appendix, the premisses are proved sequentially from left to right simply because this is the obvious and painless way of doing it. No known significance attaches to the order in which non-invertible rules are applied or formulas used in the implementations in the Appendix except where stated. Note, however, that some of the modifications of the basic algorithm described in the following impose further restrictions.

That the basic algorithm always terminates is clear, and by the contraction lemma and invertibility lemma the algorithm is sound.

6 Improving the basic algorithm

The improvements of the basic procedure to be considered in the following are directed towards speeding it up by further reducing the backtracking. What constitutes an "improvement" is not always as clear as it should be. Here the following criteria, listed in

order of importance, will be applied: i) The modification should not significantly slow down the algorithm even in special cases, ii) it should speed up the algorithm to a noticeable degree in at least some cases, iii) it shouldn't be too horribly complicated. These criteria, it will be noted, presuppose that we want to produce a general algorithm for deciding validity in intuitionistic propositional logic. (i) requires some clarification. A certain overhead must be expected when complications are introduced, and thus a slowing down of the algorithm in many cases. An improvement should not, however, be inherently bad (apart from the unavoidable overhead) for particular inputs, or cancel the effect of other improvements: we want a good general algorithm. An "improvement" which satisfies only the first criterion is not excluded, although we can derive from it only the satisfaction of knowing that unnecessary computations are avoided.

Whether the criteria are satisfied is frequently moot. One reason for this is that dramatic differences between algorithms in a particular case may be due, not to the perceived difference between the algorithms, but to a fortuitous difference at a deep level in the use of implications.

The non-invertibility of the rules $\supset\rightarrow$ and $\rightarrow\vee$ is what stands in the way of a deterministic construction of a proof. That $\supset\rightarrow$ is non-invertible has two consequences. First there is the need for contraction. I have found no way of essentially improving the way contraction is handled in the basic algorithm. The second consequence is that the order in which we use implications in the antecedent can be crucial.[4] One way of reducing the amount of computation generated by implications in the antecedent would be to introduce principles for intelligently selecting an implication to use. No doubt this is a necessary step if an efficient complete algorithm is to be found, since, in simple and naturally occurring examples, the initial choice of implication to use makes a difference by a factor of several thousand to the execution time. The trouble with implications becomes painfully apparent if one sets out to prove e.g. the formula

$$\forall x\exists y p(x,y)\&\forall x\forall y\forall z(p(x,y)\&p(y,z)\supset p(x,z))\supset\exists x p(x,x)$$

restricted to the domain {1,...n} (and translated into a propositional formula). The modifications to be considered here, however, are resolutely "mechanical" rather than "intelligent" in character. That is, implications are chosen at random and no logical or pattern-based analysis of sequents is attempted. The techniques of implication gathering, repetition elimination, and irrelevance checking introduced below are concerned with the use of implications.

The non-invertibility of $\rightarrow\vee$ is not as serious a problem. It is partly dealt with by the

[4] A simple example: a proof of the sequent $p_1\supset p_2\vee q,\ldots,p_{n-1}\supset p_n\vee q,p_n\supset q,p_1\rightarrow q$ must use the n implications in the indicated order.

or-locking technique described below.

Or-locking and repetition elimination represent the most straightforward type of optimization of the basic algorithm: a *pruning* of that algorithm. By this I mean cutting away paths of computation which can never (in the use of the algorithm at issue) lead to any proof in G. Thus in the case of or-locking, the use of $\rightarrow\vee$ is a mere waste of time in the indicated circumstances. The only way in which a pruning can slow down the algorithm is by consuming more time in checking that the necessary conditions are satisfied than is gained by avoiding unnecessary computations. Evaluating modifications which are not prunings is more difficult. By a *restriction* of an algorithm I will mean a modification which consists in cutting away some paths of computation: a pruning is thus a special kind of restriction. Using invertible rules before non-invertible rules is an example of a restriction which is not a pruning.

It must be stated that all versions of the algorithm considered here remain unfeasible as general decision procedures: n does not have to be very large for the proof of the formula above to take "forever". Also note that attempting to reduce the amount of backtracking means to try to approach the deterministic procedure for finding proofs in the classical sequent calculus, a highly inefficient exponential procedure for deciding validity in classical propositional logic. This said, it should also be noted that the algorithms presented work well over a large area and have practical utility. Experience and informal considerations suggest that intuitionistic propositional logic is inherently more computationally complex than classical, but I don't know of any formal results to this effect.[5]

7 Or-locking

In the basic algorithm, invertible rules are applied until we have a reduced sequent $\Gamma\rightarrow A$. If A is atomic, we must choose an implication to use in Γ; if A is a disjunction we also have the choice of applying $\rightarrow\vee$. In general all permutations of these rules must be tried. For example, in proving $p\supset q\rightarrow(r\supset r)\vee s$ we must apply $\rightarrow\vee$ first, in proving $p,p\supset q\vee r\rightarrow q\vee r$ we must first apply $\supset\rightarrow$. But now suppose we decide always to try one of the rules $\rightarrow\vee$, $\supset\rightarrow$ before the other. Given the choice of trying $\rightarrow\vee$ before $\supset\rightarrow$, we can improve the procedure by the technique of or-locking now to be described. Assuming that the attempt to apply $\rightarrow\vee$ to prove $\Gamma\rightarrow C\vee D$ fails, the next step is to utilize an implication in Γ ($\Gamma = A\supset B,\Delta$):

[5] Melvin Fitting informs me that R.Statman has proved intuitionistic propositional logic to be PSPACE complete.

$$\frac{A\supset B,\Delta\rightarrow A \quad B,\Delta\rightarrow C\vee D}{A\supset B,\Delta\rightarrow C\vee D}$$

Clearly there is no point, when trying to prove the right premiss, in seeking a proof of $B,\Delta\rightarrow C$ or $B,\Delta\rightarrow D$. For if such a proof exists we know that the left premiss is not provable, since $A\supset B,\Delta\rightarrow C$ and $A\supset B,\Delta\rightarrow D$ are not provable. This argument can be continued upward in a branch as long as $\vee\rightarrow$ is not applied. In more formal terms:

Or-locking lemma: Suppose $\Gamma\rightarrow C$ and $\Gamma\rightarrow D$ are not provable. Then there is no proof of $\Gamma\rightarrow C\vee D$ in which a branch starting with the bottom sequent leads to an application of $\rightarrow\vee$ without any intervening switches or applications of $\vee\rightarrow$.

Proof: Suppose there is such a proof. A simple induction shows that $\Gamma\rightarrow C$ or $\Gamma\rightarrow D$ is provable. ■

Hence the use of or-locking: if the attempt to prove a reduced sequent $\Gamma\rightarrow C\vee D$ using $\rightarrow\vee$ fails, the disjunction $C\vee D$ is locked, i.e. no further application of $\rightarrow\vee$ is allowed, until the next switch or application of $\vee\rightarrow$. Note that $\Delta\rightarrow C$ or $\Delta\rightarrow D$ may well be provable even though $C\vee D$ is locked in $\Delta\rightarrow C\vee D$, but proofs of those sequents can yield no proof of the parent sequent $\Gamma\rightarrow C\vee D$.

There is the possibility of a further refinement: Suppose we come to an application of $\vee\rightarrow$ to the sequent $A\vee B,\Delta\rightarrow C\vee D$ and the disjunction $C\vee D$ is locked. If we now (unlocking the disjunction) succeed in proving $A,\Delta\rightarrow C$ we need not try $B,\Delta\rightarrow C$; if $A,\Delta\rightarrow D$ succeeds, we need not try $B,\Delta\rightarrow D$. Unfortunately, I have not been able to establish that this refinement has practical significance. Or-locking as described above, however, yields significant returns in many cases and entails little overhead. Only a simple flag is needed to implement or-locking in the algorithm.

8 Repetition elimination

Suppose that in seeking to prove $\Gamma,A\supset B,\Gamma',\Delta\rightarrow C$, where Γ,Γ' contain implications and Δ atomic formulas, we choose $A\supset B$ as the implication to use (having previously used the implications in Γ and failed to prove the sequent), succeed in proving $\Gamma,B,\Gamma',\Delta\rightarrow C$ and then tackle $\Gamma,[A\supset B],\Gamma',\Delta\rightarrow A$, the brackets indicating that $A\supset B$ is the provisional implication, not to be used again until a new component of A has been transferred. Now

suppose this leads (via applications of $\rightarrow\&$, $\rightarrow\vee$, $\rightarrow\supset$) to a sequent $\Gamma,[A\supset B],\Gamma',\Delta\rightarrow D$, no new formulas having been transferred, and we come to a new application of $\supset\rightarrow$. At this point we can safely ignore every implication $E\supset F$ in Γ, since we have already tried to prove $\Gamma,A\supset B,\Gamma',\Delta\rightarrow E$ and failed. To include repetition elimination in the algorithm is to look out for this situation and go directly to the implications in Γ' when it arises.

Repetition elimination carries with it a somewhat greater computational overhead than or-locking in the need to keep track of the position of the provisional implication. Experiments suggest that repetition elimination has some value but does not in any case dramatically decrease execution times.

9 Combining implications

The third modification of the basic algorithm to be considered consists in rewriting sequents by combining implications. There are two obvious transformations of this kind: replacing $A\supset B,A\supset C$ by $A\supset B\&C$ (&-gathering) and replacing $A\supset C,B\supset C$ by $A\vee B\supset C$ ($\vee$-gathering). Formally, the validity of these transformations in checking provability in G is established via cut elimination or the Kripke completeness of G.

&-gathering and $\vee$-gathering both improve the procedure by reducing the number of implications and eliminating redundancies. &-gathering is particularly beneficial in that the components of A are extracted, after &-gathering, in only one branch of the attempted proof. In the case of $\vee$-gathering a further modification must be made to avoid introducing unnecessary computations: Suppose we apply $\vee$-gathering to $A\supset C,B\supset C$ in $A\supset C,B\supset C,\Gamma\rightarrow D$ and then go on to use $A\vee B\supset C$. In proving the left premiss we may clearly in this situation presuppose that $\rightarrow\vee$ is applicable, i.e. only try proving $A\vee B\supset C,\Gamma\rightarrow A$ or $A\vee B\supset C,\Gamma\rightarrow B$.

&-gathering and $\vee$-gathering can be formulated without any rewriting of sequents. To use &-gathering is to apply the rule

$$\frac{[A\supset B_1,..A\supset B_n],\Gamma\rightarrow A \qquad B_1,..B_n,\Gamma\rightarrow C}{A\supset B_1,..A\supset B_n,\Gamma\rightarrow C}$$

where the brackets indicate that none of the implications $A\supset B_i$ is to be used again until a new component has been transferred from A. To rewrite the sequent using & is a convenient way of collecting the implications at an early stage and not having to look

through the list of implications when the rule is to be applied.

Similarly $\vee$-gathering essentially consists in applying the rule

$$\frac{[A_1 \supset B, .. A_n \supset B], \Gamma \rightarrow A_i \qquad B, \Gamma \rightarrow C}{A_1 \supset B, .. A_n \supset B, \Gamma \rightarrow C}$$

The simplest way of implementing this rule is to introduce a special connective split-$\vee$ for $\vee$-gathering, together with the rule that $\Gamma \rightarrow A$ split-$\vee$ B can be proved only by proving $\Gamma \rightarrow A$ or $\Gamma \rightarrow B$.

10 Irrelevance checking

In proving $A \supset B, \Gamma \rightarrow C$ it may well happen that we prove $B, \Gamma \rightarrow C$ without using B in any essential way. Thereupon we may spend a lot of time trying to prove $A \supset B, \Gamma \rightarrow A$. If this fails, we try the next implication in Γ. Clearly there is scope here for a radical improvement of the procedure, viz. incorporating the observation that since $B, \Gamma \rightarrow C$ has been proved without using B, we have already proved $\Gamma \rightarrow C$.

The notion of "essential use" calls for an explanation. This notion presupposes that we keep track of formulas in a proof, so that the formula introduced in each application of an inference rule is uniquely identifiable (as a particular occurrence of a formula), as are the formula occurrences yielding that conclusion, and also the formula occurrences in virtue of which a sequent is seen to be an axiom. That is, it presupposes an occurrence analysis of the proof. We make *essential* use of a formula B in a proof of $B, \Gamma \rightarrow C$ if some axiom in the proof is an axiom in virtue of a formula occurrence related to the indicated occurrence of B. If no essential use is made of B, we get a proof of $\Gamma \rightarrow C$ by just deleting B and every formula occurrence related to B.

As here explained, the notion of essential use is relative to an occurrence analysis of a proof. We could explain it in a non-relative way by saying that the indicated occurrence of B in $B, \Gamma \rightarrow C$ is non-essential in a proof of the sequent if there exists some occurrence analysis relative to which B is not essential. Then it still holds that $\Gamma \rightarrow C$ is provable if $B, \Gamma \rightarrow C$ has a proof in which B is not used essentially. To verify this latter property entails, however, going through all possible occurrence analyses of the proof. We don't want to do this. Instead the algorithm inspects only the one occurrence analysis which is

yielded by the sequence of operations by which a proof happens to be constructed in an application of the algorithm.

Experiments indicate that irrelevance checking does indeed make a dramatic difference to execution times (by a factor of 100) when there are many or complicated irrelevant premisses. Note, however, that this technique does in general detect semantical irrelevance. That is, even if $\Gamma \rightarrow C$ is in fact valid, we cannot expect in general that our proof of $B,\Gamma \rightarrow C$ will make no essential use of B. Consider for example the awful formula in §6. In the case n=3, the propositional formula generated by (automatically) transforming the predicate logic formula contains 27 implications. In the proof of this formula there are 27 cases (generated by $\vee\rightarrow$), each of which must tackle the 27 implications. In each case we need at most two of the implications. Consider e.g. the case where we have assumed p(a1,a2),p(a2,a1),p(a3,a1). Here we need only one implication to conclude that p(a1,a1), and $p(a3,a3)\&p(a3,a3) \supset p(a3,a3)$ is irrelevant. If we begin with this latter implication, however, we will find an essential use for p(a3,a3), e.g. in a derivation of the antecedent of $p(a3,a3)\&p(a3,a1) \supset p(a3,a1)$, another irrelevant implication.

Perhaps the simplest way of incorporating irrelevance checking in the algorithm is to reduce occurrences to formulas by a preliminary transformation of formulas in which every occurrence of an atomic formula is rendered unique by adding an index. Thus the various occurrences of p become at(p,0),at(p,1),.. We can now say that a formula D is used in an essential way in a proof if and only if a subformula of D is used at some point to establish that a sequent is an axiom. Note that in checking for axioms and at a few other points we must now check whether A and B are variants, in the sense that A is obtainable from B by substituting at(p,i) for at(p,j) (for some set of p,i,j).

References

[1] D.van Dalen
"Intuitionistic logic", in *Handbook of Philosophical Logic,* vol III, D.Reidel Publishing Company, 1986.

[2] M.Dummett
Elements of Intuitionism, Oxford 1977.

[3] T.Franzén
"Logic programming and the intuitionistic sequent calculus" Research report, SICS, 1988.

APPENDIX

The appendix contains timings for five versions of the algorithm, the test file containing the examples used, and source code in Quintus Prolog for two versions. Icalc1 is the basic version, and icalc12 contains all the modifications presented in the text, with the exception of repetition elimination.

this is icalc1 (basic)

```
prov(1)      no      717
prov(2)      yes      33
prov(3)      yes      17
prov(4)      yes      17
prov(5)      no      16
prov(6)      yes      17
prov(7)      yes      83
prov(8)      yes      0
prov(9)      yes      33
prov(10)      no      17
prov(11)      yes      17
prov(12)      yes      2300
prov(13)      yes      134
prov(14)      no      34
prov(15)      no      33
prov(16)      yes      12150
prov(17)      yes      0
prov(18)      no      117
prov(19)      yes      33
prov(20)      yes      300
```

this is icalc1 (basic)

```
con(15)      yes      617
con2(5)      yes      1867
con2(6)      yes      12233
contrans(2)      yes      1583
contrans(3)      yes      63283
prob(1)      yes      200
prob(2)      yes      72617
seq(2)      yes      1250
bigiff      yes      900
nixiff      yes      26133
chewiff      no      38850
```

this is icalc2 (or-lock)

prov(1)	no	700
prov(2)	yes	33
prov(3)	yes	0
prov(4)	yes	17
prov(5)	no	16
prov(6)	yes	17
prov(7)	yes	100
prov(8)	yes	0
prov(9)	yes	0
prov(10)	no	17
prov(11)	yes	66
prov(12)	yes	2434
prov(13)	yes	100
prov(14)	no	0
prov(15)	no	33
prov(16)	yes	7434
prov(17)	yes	0
prov(18)	no	117
prov(19)	yes	50
prov(20)	yes	283

this is icalc2 (or-lock)

con(15)	yes	617
con2(5)	yes	2116
con2(6)	yes	13017
contrans(2)	yes	667
contrans(3)	yes	8300
prob(1)	yes	216
prob(2)	yes	76117
seq(2)	yes	1067
bigiff	yes	933
nixiff	yes	26550
chewiff	no	40650

```
this is icalc3 (or-lock;&gather)

prov(1)      no      84
prov(2)      yes      17
prov(3)      yes      33
prov(4)      yes      17
prov(5)      no      17
prov(6)      yes      16
prov(7)      yes      117
prov(8)      yes      33
prov(9)      yes      17
prov(10)      no      33
prov(11)      yes      50
prov(12)      yes      2617
prov(13)      yes      117
prov(14)      no      0
prov(15)      no      33
prov(16)      yes      2050
prov(17)      yes      0
prov(18)      no      134
prov(19)      yes      16
prov(20)      yes      283

this is icalc3 (or-lock;&gather)

con(15)      yes      650
con2(5)      yes      2100
con2(6)      yes      13117
contrans(2)      yes      667
contrans(3)      yes      8733
prob(1)      yes      233
prob(2)      yes      76550
seq(2)      yes      1000
bigiff      yes      1050
nixiff      yes      9366
chewiff      no      1550
```

this is icalc11 (or-lock;&gather;new-vgather)

```
prov(1)      no      33
prov(2)      yes      17
prov(3)      yes      33
prov(4)      yes      0
prov(5)      no      0
prov(6)      yes      0
prov(7)      yes      100
prov(8)      yes      0
prov(9)      yes      0
prov(10)      no      50
prov(11)      yes      67
prov(12)      yes      2933
prov(13)      yes      50
prov(14)      no      17
prov(15)      no      33
prov(16)      yes      800
prov(17)      yes      50
prov(18)      no      67
prov(19)      yes      17
prov(20)      yes      350
```

this is icalc11 (or-lock;&gather;new-vgather)

```
con(15)      yes      650
con2(5)      yes      2117
con2(6)      yes      13766
contrans(2)      yes      767
contrans(3)      yes      9100
prob(1)      yes      167
prob(2)      yes      3783
seq(2)      yes      750
seq(3)      yes      218100
bigiff      yes      5650
nixiff      yes      8467
chewiff      no      600
irrelevancetest      yes      591467
```

this is icalc12 (or-lock;&gather;using;newvgather)

```
prov(1)      no      133
prov(2)      yes      100
prov(3)      yes      67
prov(4)      yes      34
prov(5)      no      67
prov(6)      yes      83
prov(7)      yes      117
prov(8)      yes      166
prov(9)      yes      100
prov(10)      no      300
prov(11)      yes      67
prov(12)      yes      234
prov(13)      yes      183
prov(14)      no      83
prov(15)      no      183
prov(16)      yes      1250
prov(17)      yes      83
prov(18)      no      250
prov(19)      yes      117
prov(20)      yes      733
```

this is icalc12 (or-lock;&gather;using;newvgather)

```
con(15)      yes      1350
con2(5)      yes      3516
con2(6)      yes      43467
contrans(2)      yes      1200
contrans(3)      yes      11716
prob(1)      yes      300
prob(2)      yes      5733
seq(2)      yes      1217
seq(3)      yes      51816
bigiff      yes      583
nixiff      yes      16400
chewiff      no      950
irrelevancetest      yes      1600
```

```
   /* time(X) - give time in ms for X to succeed or fail */

time(X) :- statistics(runtime,L),X,statistics(runtime,[A,B]),write(B).
time(X) :- statistics(runtime,[A,B]),write(B),fail.

  /* testfile(File,Probs) - write a table of timings of the goals in the
     list Probs to file File. */

testfile(File,Probs) :- tell(File),nl,nl,ipversion,do(Probs),told.

do([F|L]) :- nl,write(F),tab(5),xtime(F),do(L).
do([]).

xtime(F) :- statistics(runtime,L),F,statistics(runtime,[A,B]),
            write(yes),tab(5),write(B).
xtime(F) :- statistics(runtime,[A,B]),write(no),tab(5),write(B).


  /* provlist(File,N) - write a table of timings of goals prov(1) to
     prov(N) to File. */

ae(X,[],[X]).
ae(X,[Y|L],[Y|M]) :- ae(X,L,M).

provlist(1,[prov(1)]).
provlist(N,L) :- M is N-1,provlist(M,K),ae(prov(N),K,L).

provfile(File,N) :- provlist(N,L),testfile(File,L).

  /* itrans(N,F,G) - translate the predicate logic formula F, interpreted
     over the domain {a(1),...a(n)}, to a propositional formula G. */

sublist(T,X,[FX|LX],[FT|LT]) :- sub(T,X,FX,FT),sublist(T,X,LX,LT).

sublist(T,X,[],[]).

sub(T,X,X,T) :- !.

sub(T,X,FX,FT) :- !,FX =.. [P|LX],sublist(T,X,LX,LT),FT =.. [P|LT].


konj(1,X,H,G) :- !,sub(a(1),X,H,G).

konj(N,X,H,G) :- !,M is N-1,konj(M,X,H,GM),sub(a(N),X,H,Ga),G=GM and Ga.

disj(1,X,H,G) :- !,sub(a(1),X,H,G).

disj(N,X,H,G) :- !,M is N-1,disj(M,X,H,GM),sub(a(N),X,H,Ga),G=GM or Ga.


itrans(N,F1 or F2,G1 or G2) :- !,itrans(N,F1,G1),itrans(N,F2,G2).

itrans(N,F1 and F2,G1 and G2) :- !,itrans(N,F1,G1),itrans(N,F2,G2).

itrans(N,nix F,nix G) :- !,itrans(N,F,G).

itrans(N,F1 imp F2,G1 imp G2) :- !,itrans(N,F1,G1),itrans(N,F2,G2).

itrans(N,F1 iff F2,G1 iff G2) :- !,itrans(N,F1,G1),itrans(N,F2,G2).

itrans(N,all(X,F),G) :- !,itrans(N,F,H),konj(N,X,H,G).
```

```
itrans(N,exists(X,F),G) :- !,itrans(N,F,H),disj(N,X,H,G).

itrans(N,P,P).


  /* con(N) is the standard example requiring N contractions */

con(N) :- itrans(N,nix nix ( exists(x,nix p(x)) or all(x,p(x))),F),iprove(F).

/* con2(N) also requires N contractions */

con2(N) :- itrans(N,nix nix (all(x,p(x) or nix p(x))),F),iprove(F).

/* an old examination problem  */

prob(N) :- itrans(N,nix nix ( all(x,p(x) imp exists(y,q(y) and r(x,y))) and
                    nix all(x,q(x) imp exists(y,p(y) and r(x,y))) imp
                    exists(x,q(x) and nix p(x))),F),iprove(F).


 /* contrans(N) is the transformed version of con(N), requiring no
     contractions */

contrans(N) :- itrans(N,all(x,(r(x) iff nix p(x))) imp nix nix
                  (exists(x,r(x)) or all(x,p(x))),F),iprove(F).

  /* seq(N) - if p is transitive and for all x there is a y with p(x,y), then
       p(x,x) must hold for some x (in a finite domain). A grotesque misuse
         of propositional logic. */

seq(N) :- itrans(N,all(x,exists(y,p(x,y))) and
                   all(x,all(y,all(z,p(x,y) and p(y,z) imp p(x,z)))) imp
                    exists(x,p(x,x)),F),iprove(F).


prov(1) :- iprove((a iff b iff c) iff ((a iff b) iff c)). /* no */

prov(2) :- iprove(nix(a or b or c) iff nix a and nix b and nix c). /* yes */

prov(3) :- iprove(nix nix nix nix nix nix nix nix nix a imp nix a). /* yes */

prov(4) :- iprove(a and (b or c) imp (a and b) or (a and c)).  /* yes */

prov(5) :- iprove((a imp b or c) imp (a imp b) or c).    /* no */

prov(6) :- iprove((a imp b imp c and d) imp (b imp a imp (c or f))).  /* yes */

prov(7) :- iprove((a iff b) and (a1 iff b1) imp ((a or a1) iff (b or b1))).
                                                          /* yes */

prov(8) :- iprove((q or nix q) imp (((p imp q) or p) imp q) imp q). /* yes;1
                                                                      cont */

prov(9) :- iprove( ( ((p1 imp q) or p1) and ((p2 imp q) or p2) imp q) imp q).
                                                  /* yes, 2 contr. */

prov(10) :- iprove((nix a iff nix b) iff (a iff b)).   /* no */

prov(11) :- iprove(nix nix (nix(a and b) iff nix a or nix b)). /* yes;1 cont */

prov(12) :- iprove([(a imp b) imp (a iff b),a imp c or d,(a imp c) imp c,
                    (((e imp c) imp d) imp f),e imp (c or e or f)],b imp a).
                                  /* yes; 3 irrelevant premisses */
```

```
prov(13) :- iprove((a1 and a2 imp b1 or b3 imp nix nix a1 and nix nix a2
                                        imp nix nix b1 or nix nix b3)).
                                                    /* yes */

prov(14) :- iprove((((a1 imp a2) imp a3) imp a4) imp a1 or a2 or a3 or a4).
                                                   /* no */

prov(15) :- iprove((((a1 imp a2) imp a3) imp a4) imp nix a1 or nix a2
                                                    or nix a3 or nix nix a4).
                                        /* no */

prov(16) :- iprove( (a1 iff b1) and (a2 iff b2) and (a3 iff b3) imp
                     (a1 iff (b2 or b3)) iff (b1 iff (a2 or a3))). /* yes */

prov(17) :- iprove(nix nix ((p imp q or r) imp (p imp q) or r)). /* yes;1 cont
                                                                    */

prov(18) :- iprove(nix nix p or nix nix q or nix nix r iff nix nix
                   (p or q or r)).  /* no */

prov(19) :- iprove(nix nix ((a iff (a iff b)) imp b)). /* yes, 1 contr. */

prov(20) :- iprove((p or q iff nix r) and (q iff p or r) imp p). /* yes */


bigiff :- iprove(nix nix (nix (p and q) iff ((nix p or nix q)
                                      iff r or nix r))). /* yes */

nixiff :- iprove((nix a iff nix b iff nix c) iff ((nix a iff nix b) iff
                    nix c)).  /* yes */

nixnixiff :- iprove(((nix nix a iff nix nix b) iff nix nix c) iff
                     (nix nix a iff nix nix b iff nix nix c)). /* yes */

chewiff :- iprove( (a iff b iff c) imp ( (a iff b) iff c) iff (a iff b iff c)).
                 /* no */


irrelevancetest :- iprove([p iff q or r,nix nix (p iff q iff p),s imp p,
t,t imp f,s iff (q iff nix nix (p imp s)),s iff nix (p imp q iff r)],f).
```

```
:- no_style_check(single_var).

:- op(500,xfy,imp).
:- op(500,xfy,iff).
:- op(450,xfy,and).
:- op(450,xfy,or).
:- op(425,fy,nix).



ipversion :- write('this is icalc1 (basic)'),nl.

  /* member is used only for testing */

member(A,[A|L]) :- !.
member(A,[X|L]) :- member(A,L).


add(none,L,L) :- !.
add(Prov,L,[Prov|L]).

  /* memrest(L,X,Rest) is used to generate the members X of L, with
     Rest being L with the first occurrence of X removed */

memrest([X|L],X,L).
memrest([Y|L],X,[Y|M]) :- memrest(L,X,M).


  /* redant(L,Imps,Atoms,Flags,Trans,A):
     perform all possible antecedent reductions on the sequent
     L,Imps,Atoms -> A.

     Flags is a list of flags, in this version only [Prov], where
     Prov is either "none" or the current provisional implication.

     Trans is a list of the formulas transferred so far.      */


  /* first check for axioms */

redant([A|L],Imps,Atoms,Flags,Trans,A) :- !.
redant([falsum|L],Imps,Atoms,Flags,Trans,A) :- !.

 /* reductions and defined connectives */

redant([A and B|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant([A,B|L],Imps,Atoms,Flags,Trans,C).

redant([A or B|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant([A|L],Imps,Atoms,Flags,Trans,C),
      redant([B|L],Imps,Atoms,Flags,Trans,C).

redant([A iff B|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant([(A imp B) and (B imp A)|L],Imps,Atoms,Flags,Trans,C).

redant([nix A|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant([A imp falsum|L],Imps,Atoms,Flags,Trans,C).

  /* implications into Imps, atomic formulas into Atoms */

redant([A imp B|L],Imps,Atoms,Flags,Trans,C) :- !,
     redant(L,[A imp B|Imps],Atoms,Flags,Trans,C).
```

```
redant([A|L],Imps,Atoms,Flags,Trans,C) :- !,
     redant(L,Imps,[A|Atoms],Flags,Trans,C).

  /* no more antecedent reductions, so start on C */

redant([],Imps,Atoms,Flags,Trans,C) :- !,
     redcon(Imps,Atoms,Flags,Trans,C).



  /* redcon(Imps,Atoms,Flags,Trans,A):
       reduce to a reduced sequent, then try ->or, then imp-> */


  /* first reductions and defined connectives */

redcon(Imps,Atoms,Flags,Trans,A and B) :- !,
     redcon(Imps,Atoms,Flags,Trans,A),
      !,
     redcon(Imps,Atoms,Flags,Trans,B).

redcon(Imps,Atoms,Flags,Trans,A iff B) :- !,
    redcon(Imps,Atoms,Flags,Trans,(A imp B) and (B imp A)).

redcon(Imps,Atoms,Flags,Trans,nix A) :- !,
    redcon(Imps,Atoms,Flags,Trans,A imp falsum).


redcon(Imps,Atoms,Flags,Trans,A imp B) :-
    member(A,Trans),
     !,
    redcon(Imps,Atoms,Flags,Trans,B). /* ignore A if it has been
                                         transferred before */

redcon(Imps,Atoms,[Prov],Trans,A imp B) :- !,
     add(Prov,Imps,Impsp), /* ok to use Prov again */
     redant([A],Impsp,Atoms,[none],[A|Trans],B).


  /* try ->or, check for axiom, then try imp-> */

redcon(Imps,Atoms,Flags,Trans,A or B) :-
    redcon(Imps,Atoms,Flags,Trans,A) ;
    redcon(Imps,Atoms,Flags,Trans,B).

redcon(Imps,Atoms,Flags,Trans,A) :- member(A,Atoms).

redcon(Imps,Atoms,[Prov],Trans,A) :- !,
   memrest(Imps,C imp D,Imprest),
  ( redant([D],Imprest,Atoms,[Prov],Trans,A) ->
    redcon(Imprest,Atoms,[C imp D],Trans,C)
    | (!,fail) ).


   /* top level predicates */

iprove(L,A) :- redant(L,[],[],[none],[],A).
iprove(A) :- redcon([],[],[none],[],A).
```

```
ipversion :- write('this is icalc12 (or-lock;&gather;using;newvgather)'),nl.

:- no_style_check(single_var).
:- no_style_check(multiple).

:- op(500,xfy,imp).
:- op(500,xfy,iff).
:- op(450,xfy,and).
:- op(450,xfy,or).
:- op(450,xfy,splitor).
:- op(425,fy,nix).


/* dofmla(A,Ado) - make every atomic formula unique, for the "prove using"
   algorithm */


fixfmla(nix A,NixAfix) :- !,fixfmla(A imp falsum,NixAfix).
fixfmla(A iff B,Afix iff Bfix) :- !,fixfmla(A,Afix),fixfmla(B,Bfix).
fixfmla(A imp B,Afix imp Bfix) :- !,fixfmla(A,Afix),fixfmla(B,Bfix).
fixfmla(A or B,Afix or Bfix) :- !,fixfmla(A,Afix),fixfmla(B,Bfix).
fixfmla(A and B,Afix and Bfix) :- !,fixfmla(A,Afix),fixfmla(B,Bfix).
fixfmla(A,at(A,N)) :- !,getids(N,A).

/* nix must be expanded to get variants of falsum */

:- dynamic currentids/1.

currentids([]).

getids(N,A) :- currentids(L),memrest(L,at(A,M),Otherids),!,
               N is M+1,retract(currentids(L)),
               assert(currentids([at(A,N)|Otherids])).
getids(0,A) :- retract(currentids(L)),
               assert(currentids([at(A,0)|L])).

initids :- retract(currentids(L)),assert(currentids([])).

dofmla(A,Ado) :- initids,fixfmla(A,Ado).

xdolist([],[]).
xdolist([A|L],[Ado|Ldo]) :- fixfmla(A,Ado),xdolist(L,Ldo).

dolist(L,Ldo) :- initids,xdolist(L,Ldo).

/* member is used only for testing */

member(X,[X|L]) :- !.
member(X,[Y|L]) :- member(X,L).



/* same(A,B) - A and B differ only in the incarnations of atomic formulas */

same(at(A,M),at(A,N)) :- !.
same(nix A,nix B) :- !,same(A,B).
same(A1 iff A2,B1 iff B2) :- !,same(A1,B1),same(A2,B2).
same(A1 and A2,B1 and B2) :- !,same(A1,B1),same(A2,B2).
same(A1 or A2,B1 or B2) :- !,same(A1,B1),same(A2,B2).
same(A1 imp A2,B1 imp B2) :- !,same(A1,B1),same(A2,B2).
```

```
/* haspart(A,L) - a subformula of A occurs in L */


subfmla(A,A) :- !.
subfmla(A imp B,X) :- !,(subfmla(A,X);subfmla(B,X)).
subfmla(A and B,X) :- !,(subfmla(A,X);subfmla(B,X)).
subfmla(A or B,X) :- !,(subfmla(A,X);subfmla(B,X)).
subfmla(A iff B,X) :- !,(subfmla(A,X);subfmla(B,X)).
subfmla(nix A,X) :- !,subfmla(A,X).

haspart(A,[B|L]) :- subfmla(A,B),!.
haspart(A,[B|L]) :- haspart(A,L).


/* samemember(A,X,L) - X is the first element of L which is the same as A */

samemember(A,X,[X|L]) :- same(A,X),!.
samemember(A,X,[Y|L]) :- samemember(A,X,L).


append([],L,L).
append([X|M],L,[X|S]) :- append(M,L,S).


add(none,L,L) :- !.
add(Prov,L,[Prov|L]).

  /* memrest(L,X,Rest) is used to generate the members X of L, with
     Rest being L with the chosen occurrence of X removed */

memrest([X|L],X,L).
memrest([Y|L],X,[Y|M]) :- memrest(L,X,M).

/* findcons(Imps,A,X,Imprest) - X is the cons of the first imp in Imps
       with an ant same as A; Imprest is Imps with this imp removed */

findcons([A1 imp X|Imprest],A,X,Imprest) :- same(A1,A),!.
findcons([B|Imps],A,X,[B|Imprest]) :- findcons(Imps,A,X,Imprest).

/* findant(Imps,B,X,Imprest) - X is the ant of the first imp in Imps
       with a cons same as B; Imprest is Imps with this imp removed */

findant([X imp B1|Imprest],B,X,Imprest) :- same(B1,B),!.
findant([A|Imps],B,X,[A|Imprest]) :- findant(Imps,B,X,Imprest).


  /* redant(L,Imps,Atoms,Flags,Trans,A):
     perform all possible antecedent reductions on the sequent
     L,Imps,Atoms -> A.

     Flags is a list of flags, in this version [Prov,O1], where
     Prov is either "none" or the current provisional implication,
     and O1 is the or-locking flag "lock or "nolock".

     Trans is a list of the formulas transferred so far.      */


  /* first check for axioms */

redant([A,B],[A|L],Imps,Atoms,Flags,Trans,B) :- same(A,B),!.
redant([at(falsum,N)],[at(falsum,N)|L],Imps,Atoms,Flags,Trans,A) :- !.

 /* reductions and defined connectives */
```

```
redant(U,[A and B|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant(U,[A,B|L],Imps,Atoms,Flags,Trans,C).

redant(U,[A or B|L],Imps,Atoms,[Prov,O1],Trans,C) :- !,
     /* unlock C */
      redant(U1,[A|L],Imps,Atoms,[Prov,nolock],Trans,C),
      redant(U2,[B|L],Imps,Atoms,[Prov,nolock],Trans,C),
      append(U1,U2,U).

redant(U,[A iff B|L],Imps,Atoms,Flags,Trans,C) :- !,
      redant(U,[(A imp B) and (B imp A)|L],Imps,Atoms,Flags,Trans,C).

  /* implications into Imps, atomic formulas into Atoms */

  /* do &-gathering */

redant(U,[A imp B|L],Imps,Atoms,Flags,Trans,C) :-
  findcons(Imps,A,X,Imprest),
   !,
  (same(X,B) -> redant(U,L,Imps,Atoms,Flags,Trans,C) |
    redant(U,L,[A imp B and X|Imprest],Atoms,Flags,Trans,C)).

/* note: which A is used is random */

 /* do splitor-gathering */

redant(U,[A imp B|L],Imps,Atoms,Flags,Trans,C) :-
  findant(Imps,B,X,Imprest),
   !,
  (same(X,A) -> redant(U,L,Imps,Atoms,Flags,Trans,C) |
    redant(U,L,[A splitor X imp B|Imprest],Atoms,Flags,Trans,C)).

redant(U,[A imp B|L],Imps,Atoms,Flags,Trans,C) :- !,
     redant(U,L,[A imp B|Imps],Atoms,Flags,Trans,C).

redant(U,[A|L],Imps,Atoms,Flags,Trans,C) :- !,
     redant(U,L,Imps,[A|Atoms],Flags,Trans,C).

  /* no more antecedent reductions, so start on C */

redant(U,[],Imps,Atoms,Flags,Trans,C) :- !,
     redcon(U,Imps,Atoms,Flags,Trans,C).



  /* redcon(Imps,Atoms,Flags,Trans,A):
     reduce to a reduced sequent, then try ->or, then imp-> */

  /* first reductions and defined connectives */

redcon(U,Imps,Atoms,Flags,Trans,A and B) :- !,
     redcon(U1,Imps,Atoms,Flags,Trans,A),
      !,
     redcon(U2,Imps,Atoms,Flags,Trans,B),
     append(U1,U2,U).

redcon(U,Imps,Atoms,Flags,Trans,A iff B) :- !,
    redcon(U,Imps,Atoms,Flags,Trans,(A imp B) and (B imp A)).


redcon(U,Imps,Atoms,Flags,Trans,A imp B) :-
    member(A,Trans),
```

```
     !,
    redcon(U,Imps,Atoms,Flags,Trans,B). /* ignore A if it has been
                                           transferred before */

redcon(U,Imps,Atoms,[Prov,O1],Trans,A imp B) :- !,
     add(Prov,Imps,Impsp), /* ok to use Prov again */
     redant(U,[A],Impsp,Atoms,[none,O1],[A|Trans],B).


  /* try ->or provided the consequent is not locked,
                            check for axiom, then try imp-> */

redcon(U,Imps,Atoms,[Prov,nolock],Trans,A or B) :-
    redcon(U,Imps,Atoms,[Prov,nolock],Trans,A) ;
    redcon(U,Imps,Atoms,[Prov,nolock],Trans,B).

redcon(U,Imps,Atoms,Flags,Trans,A splitor B) :- !,
   (redcon(U,Imps,Atoms,Flags,Trans,A);redcon(U,Imps,Atoms,Flags,Trans,B)).

redcon([A,B],Imps,Atoms,Flags,Trans,A) :- samemember(A,B,Atoms).

  /* note: the locking is harmless if A is atomic */

redcon(U,Imps,Atoms,[Prov,O1],Trans,A) :- !,
   memrest(Imps,C imp D,Imprest),
   ( redant(U1,[D],Imprest,Atoms,[Prov,lock],Trans,A) ->
    ifused(U1,U,Imprest,Atoms,C imp D,Trans)
     | (!,fail) ).

ifused(U1,U,Imprest,Atoms,C imp D,Trans) :-
     haspart(D,U1),
     !,
     (redcon(U2,Imprest,Atoms,[C imp D,nolock],Trans,C),append(U1,U2,U)).

ifused(U1,U,Imprest,Atoms,C imp D,Trans) :- U=U1.



   /* top level predicates */

iprove(L,A) :-dolist(L,M),dofmla(A,B),redant(U,M,[],[],[none,nolock],[],B).
iprove(A) :- dofmla(A,B),redcon(U,[],[],[none,nolock],[],B).
```