Some Applications of Gentzen's Proof Theory in Automated Deduction

Michael Beeson
Mathematics and Computer Science
San Jose State University
San Jose, California 95192, USA

May 29, 1990

Reasoning and problem-solving programs must eventually allow the full use of quantifiers and sets, and have strong enough control methods to use them without combinatorial explosion. [1]

–J. McCarthy

Abstract

We show that Prolog is intimately connected with Gentzen's cut-free sequent calculus **G** , analyzing Prolog computations as the construction of certain cut-free derivations. We introduce a theorem-proving program GENTZEN based on Gentzen's sequent calculus, which incorporates some features of Prolog's computation procedure. We show that GENTZEN has the following properties: (1) It is (non-deterministically) sound and complete for first-order intuitionistic predicate calculus; (2) Its successful computations coincide with those of Prolog on the Horn clause fragment (both deterministically and non-deterministically). The proofs of (1) and (2) contain a new proof of the completeness of (non-deterministic) Prolog for Horn clause logic, based on our analysis of Prolog in terms of Gentzen sequents instead of on resolution. GENTZEN has been implemented and tested on examples including some proofs by induction in number theory, an example constructed by J. McCarthy to show the limitations of Prolog, and "Schubert's Steamroller." An extension of GENTZEN also provides a decision procedure for intuitionistic propositional calculus (but at some cost in efficiency).

[1] McCarthy [1987], p. 1032.

Contents

1 Introduction

Prolog is a programming language in which the underlying computation mechanism is logical deduction. The language uses a subset of the first-order predicate calculus called Horn clauses. These may be defined as formulae $A_1, \ldots, A_n \rightarrow B$, where the A_i and B are atomic. The deduction process used by Prolog has been explained in the literature as a form of resolution (Lloyd [1984]). It is one of our purposes here to give an alternate, and we believe more useful, explanation of Prolog's deduction process, in terms of the classical proof theoretic results of G. Gentzen. Acquaintance with the systems of Gentzen (see Kleene pp. 452-501) is presumed.

Our second purpose is to extend Prolog to the full intuitionistic first-order predicate calculus, without sacrificing efficiency. [2] Note that this is a *stronger* result than completeness for classical logic: we can define classical logic within intuitionistic logic, simply by expressing $\exists$ and $\vee$ in terms of $\forall$, $\wedge$, and $\neg$. Intuitionistic logic is *more general* than classical logic. The (family of) program(s) we provide for extending Prolog to intuitionistic predicate calculus is called GENTZEN in honor of the proof theorist G. Gentzen, whose ideas have illuminated proof theory for half a century and may yet help in automated deduction.

Both Prolog and GENTZEN are described by algorithms which can be interpreted either non-deterministically or deterministically. All known metamathematical results about Prolog apply only to the non-deterministic version,

[2] Of course, one can construct examples where even Prolog requires exponential time for backtracking. We are discussing *practical* efficiency, not theoretical efficiency.

while it is the deterministic version that executes in practice. In this paper, we prove corresponding metamathematical results for the non-deterministic version of GENTZEN.

The deterministic version of Prolog makes sacrifices of both soundness and completeness for speed. GENTZEN makes similar sacrifices, but we also show how to make these trade-offs explicit: if you are willing to accept a slower GENTZEN, you can come much closer to completeness with deterministic GENTZEN.

If the simpler versions of GENTZEN are run deterministically under Prolog, they work very well on many interesting examples. However, on examples require complicated search, such as Schubert's Steamroller, they fail rather miserably. The control structure provided in these simple versions for the use of the rules is simply inadequate. There are, however, a wide variety of possibilities for deterministic versions of GENTZEN. We believe that the framework provided by non-deterministic GENTZEN provides a unifying conceptual apparatus into which one can translate and compare various approaches to automated deduction. We have, for example, discovered simple control structures for the use of GENTZEN's rules which permit very rapid solution of the Steamroller and other such problems. This will be taken up in a later paper.

The main results of this paper can be summarized as follows:

• The completeness of (non-deterministic) Prolog for Horn clause logic is a consequence of the proof theory of a Gentzen sequent calculus. [3] A Prolog computation uses backtracking and unification to construct a Gentzen derivation of a sequent representing the program and goal.

• GENTZEN's computations on sequents representing Prolog programs and queries exactly parallel Prolog's computations. This is true both deterministically and non-deterministically.

• Non-deterministic GENTZEN is complete for intuitionistic predicate calculus.

• If deterministic GENTZEN terminates, it decides whether the input is provable in intuitionistic predicate calculus; incompleteness only can result from infinite regress.

• GENTZEN has been implemented and tested, and can rapidly prove several test problems, including some proofs by induction in number theory and an example which J. McCarthy invented to show the limitations of Prolog. It can also do many independence proofs by terminating with failure.

• If certain "redundancy checks" are added to GENTZEN, it runs somewhat slower, but then provides a decision procedure for intuitionistic propositional calculus, thus improving on Prolog even on the propositional fragment.

The analysis of Prolog given here answers two questions which haunted the author for several years after he began programming in Prolog:

[3] More precisely, of the completeness of Gentzen's cut-free rules, which can be directly proved without the cut-elimination theorem, (or via the cut-elimination); and of an additional not-quite-trivial "Permutation Lemma".

• Why is it that Prolog seems so "constructive", yet resolution is non-constructive?

• What is the true source of Prolog's efficiency?

The answers to these questions are clear and short, so we give them here, even though you may have to read the paper to understand them fully: Prolog is constructive because its computation method is essentially the construction of a proof in an intuitionistic sequent calculus. The source of its efficiency is threefold: (1) Use only cut-free intuitionistic proofs; (2) Use unification to calculate the terms required at the rules $\Rightarrow \exists$ and $\forall \Rightarrow$; and (3) never go up the right branch of a left implication; use left implication only if the right premise is an axiom.

Having understood Prolog in terms of Gentzen sequents, we were able to generalize Prolog's computation method to the full intuitionistic predicate calculus. We think we have thus contributed some answers to the following two general questions:

• How can logic programming be extended to first-order predicate calculus?

• What general logical principles (as opposed to domain-specific knowledge) can guide the construction of improved automated theorem-provers?

Namely, the same three principles just mentioned, with (3) modified to permit going up the right branch of a left implication only if all other proof methods fail.

GENTZEN bridges the gap between logic programming and automatic theorem proving. If you add the clause `derive(( Gamma => A)):- call(A)` to the top of the program listing for GENTZEN, an attempt to prove A can call on Prolog's built-in procedures, so that what you have is an extension of Prolog to full first-order predicate calculus, but able to access all the non-logical features of Prolog at the same time.

Although GENTZEN can be viewed as an extension of Prolog, we think in addition that the framework of GENTZEN is a good skeleton on which to build a general-purpose mathematical theorem prover. Personally we find Prolog adequate as a programming language, and our interest in GENTZEN is rather from the perspective of automated deduction. GENTZEN makes theorem-proving look like logic programming: In order to get a proof, you pay attention to the order in which you list your axioms, thus controlling the execution of the prover. There is a trade-off between efficiency and completeness. Since GENTZEN is non-deterministically complete, there are no obvious theoretical limits to its deterministic abilities. For example, if you were willing to pay the price of replacing its Prolog-like search strategy by "depth-first iterative deepening", you could achieve deterministic completeness. Bledsoe [1984] (p. 93) says:

> Experience so far has shown that complete procedures tend to be weak, in the sense that they take too long to prove easy theorems (or cannot prove them at all) ...

With GENTZEN's basic architecture, we can have either a fast incomplete prover or a slower complete prover, and we can explore many possible intermediate versions. The version presented here is the fast incomplete version, which we show is still more than adequate as a useful extension of Prolog to problems involving quantifiers.

The completeness of non-deterministic GENTZEN makes deterministic GENTZEN "partially complete": when it terminates, it decides provability of the input, so it can be used for independence proofs. (Note that this means independence from intuitionistic axiom systems, which is often rather difficult to prove.)

Finally, we would like to mention a widespread misconception: that intuitionistic logic is esoteric and not useful for mainstream mathematics. This is really two misconceptions: one, that using intuitionistic logic involves rejecting classical logic. This is not the case, as explained above. Rather, it involves *refining* classical logic to keep track of extra information (about algorithms) provided in the proof. The other misconception is related: that constructive methods can't be powerful in mathematics. On the contrary, the more information you keep track of (without getting lost) the better off you are. We think this principle should apply in automated deduction algorithms as well as to people, and so we think more work should be done on proving theorems constructively by machine. We hope GENTZEN will contribute to this program. We have purposely not considered versions of GENTZEN based on classical sequent calculi, believing that restriction to the intuitionistic rules should improve performance.[4]

We are under no illusions that four pages of Prolog code make a general purpose theorem prover. Rather, we think GENTZEN offers a particularly fruitful underlying architecture for a theorem-prover. There are natural places to integrate the following into GENTZEN: type theory, equality reasoning using symbolic computation, semantic methods, symbolic computation, control of expansion or non-expansion of definitions. [5] Whether or not our hopes for the power of theorem-provers based on GENTZEN's architecture will be borne out remains to be seen; the proof of the pudding lies in actually building such theorem-provers and using them to prove theorems. It is thus not the point of this paper to discuss "real control structures and efficiency in a practical sense", in the words of a referee. On the contrary, the purpose of this paper is to exhibit a unified theoretical framework based on the sequent calculus, in which one can understand both Prolog and general-purpose theorem-provers, and in which one has the flexibility to use the logical form of both goals and axioms to specify control structures.

Prerequisites to this paper are an acquaintance with Gentzen's logical systems and an acquaintance with unification, up through the concept of "most general unifier". For background on Gentzen, see Kleene [1952], pp. 481-501.

[4] When considering devices to improve completeness at the expense of efficiency, the classical sequent calculi should be reconsidered.

[5] See also the work of N. Shankar, described in Section 11, for an improvement to GENTZEN's purely logical search algorithm.

For background on unification, see Lloyd [1984]. Familiarity with Prolog will be required to actually read the program of GENTZEN, but in theory it is not required to read the paper.

A detailed comparison of our work with that of Bledsoe, Bowen, Boyer and Moore, Constable, Feferman, Felty, Hayashi, Lifschitz, McCarty, Miller, Paulson, Shankar, Stickel, and others is given in the last section of the paper.

2 Proof-theoretic Preliminaries

Matters of Notation

The basic reference we use for sequent calculi is Kleene [1952]. We have, however, changed the notation for sequents slightly, writing $\rightarrow$ for implication and $\Rightarrow$ for the separator between the left part (antecedent) and the right part (succedent) of a sequent.

We have also adopted a different typography for proof trees than is traditional. Instead of drawing tree diagrams growing upwards from goal to axioms, our trees grow left-to-right, with indentation being used instead of horizontal bars to indicate passage to subtrees. For example, the rule $\rightarrow\Rightarrow$ is written in traditional notation as

$$\frac{A \rightarrow B, \Gamma \Rightarrow A \qquad\qquad B, A \rightarrow B, \Gamma \Rightarrow \Theta}{A \rightarrow B, \Gamma \Rightarrow \Theta}$$

and in our notation as

$$\begin{aligned} A \rightarrow B, \Gamma &\Rightarrow \Theta \\ A \rightarrow B, \Gamma &\Rightarrow A \\ B, A \rightarrow B, \Gamma &\Rightarrow \Theta \end{aligned}$$

This notation is similar to that used in the "proof refinement logic" of NuPrl (Constable *et. al.* [1986]). It is much more convenient to write or typeset than the traditional notation. However, it is almost impossible to break the habit of visualizing proof trees with the axioms at the top and the conclusions at the bottom; so we will retain the informal meaning of 'above' as meaning nearer the leaves of the tree (the axioms) and 'below' as meaning nearer the root, even though 'above' means 'to the right and below on the page' in the newer notation.

Let me remind the reader that there are several different Gentzen sequent calculi. **G3** (Kleene [1952], p. 481) is the one designed to provide a decision procedure for intuitionistic propositional calculus. In this calculus, as you work up the tree from the goal, you do not discard formulae; this enables a bound (in the propositional case) on the size of the proof tree.

G1 (Kleene [1952], p. 442) does not provide a propositional decision procedure, because there is a rule that permits dropping duplicated hypotheses. When that rule is used in reverse, you lose control over the possible size of the proof. On the other hand, the rule $\rightarrow\Rightarrow$ in **G1** is closer to Prolog than **G3**.

Note: In Kleene [1952], **G1** and **G3** contain the cut rule. We use these letters instead to stand for the corresponding *cut-free* systems. *All systems in this paper are cut-free.*

In a Gentzen sequent $\Gamma \Rightarrow \Delta$, usually Γ and Δ are finite sets (or lists) of formulae. The difference between intuitionistic and classical Gentzen calculi is only that in the intuitionistic calculus, the succedent Δ is allowed to contain at most one formula (it may be empty). If we permit the atomic formula `false` to stand for the empty sequent in the succedent, we can assume Δ always contains exactly one formula, in which case we may as well write sequents in the form $\Gamma \Rightarrow \Theta$, where Θ is a formula and Γ is a finite set of formulae.[6] The calculus **G3** as formulated in Kleene in effect treats Γ as a set, i.e. duplicates and order are ignored, while **G1** treats Γ as a list. The treatment as a set is more convenient for our purposes, even though ultimately sets must be represented as lists for an implementation.

The Sequent Calculi **G4** *and* **G**

In order to explain Prolog, we found it necessary to develop a hybrid version of Gentzen's calculus incorporating some of the features of **G1** and some of the features of **G3**. **G3** differs from **G1** by carrying more formulae from conclusion to hypothesis. For our purposes, the definition of **G3** was "overkill". It isn't necessary to carry quite so many formulae to the hypothesis. [7] Our system **G** is an extension of **G3**. It is similar to **G3**, but permits carrying fewer formulae to the hypothesis under certain circumstances. This is accomplished by giving **G** a different rule $\rightarrow\Rightarrow$ (and also $\wedge \Rightarrow$ and $\vee \Rightarrow$) than **G3**. A related system is **G4**, which *requires* the omission of premisses whenever **G** *permits* it. We shall refer to a **G4** derivation as a *strict* **G** derivation.

In rule $\rightarrow\Rightarrow$, the question is whether the principal formula $A \rightarrow B$ is to occur in the premise(s) as well as the conclusion of the rule. In **G3** it does, in **G1** it does not. Our system **G** adopts the following rule: In case $A- > B$ is obtained by a substitution from the matrix of a universally quantified member of Γ (the rest of the antecedent), then it *may* be omitted from the premise of the rule. Otherwise, it shall occur. The universally quantified member of Γ may have several universal quantifiers. Here are the two rules for comparison:

$$\frac{A \rightarrow B, \Gamma \Rightarrow C}{A \rightarrow B, \Gamma \Rightarrow A} \quad \textbf{G3}$$

[6] Unless `false` is specified "by the user" in listing the axioms (the formulae of the antecedent), it can never occur except in the succedent, where it can be interpreted as the empty list. However, if the user is permitted to use `false`, as seems natural, an argument which we have not given is required to show that the rules of the system handle it correctly. To stay strictly in the scope of proved metatheorems, `false` should not be used in axioms; use $\neg$ instead.

[7] In fact, there is a version of GENTZEN based on each of these calculi. It is only the **G** and **G4** versions, however, which exactly imitate Prolog. The others work in a slightly less efficient way. In fact, on certain computations the **G3** version suffers practical problems due to Prolog's lack of an "occurs check" in unification, which the **G** and **G4** versions do not seem to. However, on all the examples in this paper, all of these versions run more or less equally well.

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

That rule applies in **G** too; but when $A \to B$ is obtained by a substitution from the matrix of a universally quantified member of Γ, the following rule can be applied instead:

$$\begin{array}{rl} A \to B, \Gamma \Rightarrow C & \mathbf{G} \\ \Gamma \Rightarrow A & \\ B, \Gamma \Rightarrow C & \end{array}$$

The rules $\wedge \Rightarrow$ and $\vee \Rightarrow$ are also changed in **G**. In **G3**, the rule $\wedge \Rightarrow$ causes branching: there are two rules with the same conclusion. There is no reason (except possibly symmetry with rule $\Rightarrow \vee$) for this; we can just take this one rule instead:

$$\begin{array}{c} A \wedge B, \Gamma \Rightarrow C \\ A, B, \Gamma \Rightarrow C \end{array}$$

Similarly, in rule $\vee \Rightarrow$, there is no reason to carry $A \vee B$ from conclusion to premise, as either disjunct is stronger. Just take this rule instead:

$$\begin{array}{c} A \vee B, \Gamma \Rightarrow C \\ A, \Gamma \Rightarrow C \\ B, \Gamma \Rightarrow C \end{array}$$

A similar variation is made in the rules $\wedge \Rightarrow$ and $\vee \Rightarrow$.

G also admits a more general syntax than the Gentzen calculi described in Kleene [1952]. In particular, we admit formulas $\forall x A$ and $\exists x A$ where x is not a single variable, but a list of variables. In this case, however, we require that the list x be in alphabetical order. [8] There are obvious generalizations of the quantifier rules to list quantifiers. In the case of rule $\forall \Rightarrow$, adding the generalized rule is not just a matter of abbreviation: if we were to "strip off" the quantifiers one at a time (using the rule of **G3** in reverse) we would get a lot of extra formulas in the antecedent. It is important to be able to strip them all off at once.

[8] Kleene is not specific about which symbols are legal variables, specifying only that there is a fixed list of variables. We specify that the variables are any legal Prolog atoms (except `false` and `true`), and that the order on the variables is the lexicographical order determined by the Prolog system's ordering predicate. Requiring quantified lists to be in order is done for the sake of efficiency.

For reference, here are the rules exactly:

Rules of G

$$C, \Gamma \Rightarrow C \qquad (\texttt{axiom})$$

$$\frac{\Gamma \Rightarrow A \to B \qquad (\Rightarrow\to)}{A, \Gamma \Rightarrow B}$$

$$\frac{\Gamma \Rightarrow A \wedge B \qquad (\Rightarrow \wedge)}{\begin{array}{c}\Gamma \Rightarrow A \\ \Gamma \Rightarrow B\end{array}}$$

$$\frac{\Gamma \Rightarrow A \vee B \qquad (\Rightarrow \vee)}{\Gamma \Rightarrow A}$$

$$\frac{\Gamma \Rightarrow A \vee B \qquad (\Rightarrow \vee)}{\Gamma \Rightarrow B}$$

$$\frac{\Gamma \Rightarrow \neg A \qquad (\Rightarrow \neg)}{A, \Gamma \Rightarrow \texttt{false}}$$

$$\frac{A \to B, \Gamma \Rightarrow C \qquad (\to\Rightarrow)}{\begin{array}{c}(A \to B), \Gamma \Rightarrow A \\ B, (A \to B), \Gamma \Rightarrow C\end{array}}$$

where the formulae in parentheses may be omitted if $A \to B$ is a substitution instance of the matrix of a universally quantified formula in Γ.

$$\frac{A \wedge B, \Gamma \Rightarrow C \qquad (\wedge \Rightarrow)}{A, B, \Gamma \Rightarrow C}$$

$$\frac{A \vee B, \Gamma \Rightarrow C}{A, \Gamma \Rightarrow C \quad B, \Gamma \Rightarrow C} \quad (\vee \Rightarrow)$$

$$\frac{\neg A, \Gamma \Rightarrow C}{\neg A, \Gamma \Rightarrow A} \quad (\neg \Rightarrow)$$

$$\frac{\Gamma \Rightarrow \forall x A}{\Gamma \Rightarrow A[y/x] \ (y \text{ not free in } \Gamma, \forall x A)} \quad (\Rightarrow \forall)$$

$$\frac{\exists x A, \Gamma \Rightarrow C}{A[y/x], \Gamma \Rightarrow C \ (y \text{ not free in } \Gamma, C)} \quad (\exists \Rightarrow)$$

$$\frac{\forall x A, \Gamma \Rightarrow C}{A[t/x], \forall x A, \Gamma \Rightarrow C} \quad (\forall \Rightarrow)$$

$$\frac{\Gamma \Rightarrow \exists x A}{\Gamma \Rightarrow A[t/x]} \quad (\Rightarrow \exists)$$

Lemma 1 *Every derivation in* **G** *can be converted to a derivation in* **G3** *(first replacing list quantifiers by iterated quantifiers). Conversely, every derivation in* **G3** *can be converted to a strict* **G** *derivation. Consequently, the cut-elimination theorem holds for* **G** *and* **G4** *as well as for* **G3**.

Proof. ($\Rightarrow$) By induction on the length of derivations in **G**. Suppose the last inference is by rule $\rightarrow\Rightarrow$, and we are in the case where the principal formula is omitted from the hypothesis. Then by inserting some extra applications of rule $\forall \Rightarrow$, we can recover the omitted formula; indeed it was this possibility that determined the cases in which we could afford to omit the formula.

Suppose that the last inference is by rule $\wedge \Rightarrow$; let the last two lines of the derivation be

$$\frac{A \wedge B, \Gamma \Rightarrow C}{A, B, \Gamma \Rightarrow C} \quad (\wedge \Rightarrow)$$

By induction hypothesis, there is a **G3** derivation of $A, B, \Gamma \Rightarrow C$. Thus all we need to know is that **G3** is closed under the $\wedge \Rightarrow$ rule of **G**. We can prove this directly with two applications of cut elimination: start with a derivation of $A \wedge B \Rightarrow A$ and one of $A, B, \Gamma \Rightarrow C$. Apply cut-elimination with cut-formula A, to get $A \wedge B, B, \Gamma \Rightarrow C$. Take a derivation of $A \wedge B \Rightarrow B$, and use B for a cut-formula, getting $A \wedge B, \Gamma \Rightarrow C$.

Suppose the last inference is by rule $\vee \Rightarrow$; let the last two lines of the derivation be

$$\begin{array}{rl} A \vee B, \Gamma \Rightarrow C & \vee \Rightarrow \\ A, \Gamma \Rightarrow C & \\ B, \Gamma \Rightarrow C & \end{array}$$

By induction hypothesis, there are **G3** derivations of the premises (so again all we really need is the closure of **G3** under this rule, which is the same as the $\vee \Rightarrow$ rule of **G1**). Just add $A \vee B$ to the antecedent of every sequent in the derivations of the premises. The derivation will remain a derivation, unless there are free variables in $A \vee B$ that cause a violation of the "restriction on variables" above this point. This problem can be avoided by renaming eigenvariables in the derivation of $A, \Gamma \Rightarrow C$ to avoid the free variables of B, and vice-versa renaming eivenvariables in the derivation of $B, \Gamma \Rightarrow C$ to avoid the free variables of A.

Suppose that the last inference is by rule $\forall \Rightarrow$, applied to a list quantifier. If we had used the $\forall \Rightarrow$ rule of **G3** several times (one quantifier at a time) (proceeding backwards from the root), we would have produced a sequent just like the hypothesis of this inference, except with some extra formulae in the antecedent. We can add these formulae to the antecedents of every formula above this point in the proof tree. The free variables of these intermediate formulae already occur in the antecedent, so no inferences are invalidated by violations of the "restriction on variables" due to these added formulae in the antecedents.

($\Rightarrow$) By induction on **G3** derivations. Suppose the last inference of the given **G3** derivation is by rule $\rightarrow\Rightarrow$ in the case where the principal formula is not dropped in the hypothesis. Then the deduction is already in **G3**. In the case where the principal formula is dropped, we copy the principal formula to the antecedents everywhere above, first renaming eigenvariables if necessary to avoid violating the restriction on variables. Similarly, if the last inference is by rule $\wedge \Rightarrow$ or rule $\vee \Rightarrow$, we copy the principal formula to the antecedents above. That completes the proof.

Substitutions and Unification

A *substitution* is a (partial) map from terms to variables which commutes with all function symbols. Lower-case Greek letters are used to denote substitutions. They are written on the right in algebraic notation: $t\theta$ instead of $\theta(t)$. Juxtaposition of symbols for substitutions indicates composition. We write

$\theta \geq \delta$ ("θ is more general than δ") if for some μ we have $\theta\mu = \delta$. Note μ might be the identity. We regard all substitutions as defined everywhere, extending them to be the identity where they are not explicitly defined. Substitutions determined by their action on a finite number of variables can be indicated by a list of equations such as $x = a^2, y = a^3$. Similarly, if θ is a substitution satisfying $x\theta = x$, we denote by $\theta, x = t$ the new substitution which agrees with θ except at x, and takes x to the value t. This is not a "union" since we regard all substitutions as defined everywhere.

Note that substitutions can be applied to formulas as well as to terms, the difference being that formulas may contain bound variables. In this case the application of a substitution θ to a formula (or other expression with bound variables) is presumed not to affect the bound variables, even if θ has a non-trivial action on those bound variables.

A substitution θ is said to *unify* terms t and s if $t\theta$ and $s\theta$ are identical. A *most general unifier* of t and s is a unifier more general than any other unifier. The most general unifier is unique up to renamings of variables.

The Permutation Lemma

Note: The reader is invited to postpone reading the rest of this section until it is clear why it is needed.

We shall make use of the "permutation lemmas" about Gentzen's calculus. These lemmas, familiar (at least in outline) to experts in proof theory but less well-known than the material in Kleene's book, show how the order of application of Gentzen's rules can be changed, so that one may assume that a cut-free proof of a given sequent ends with a certain rule. This restructuring of the proof tree cannot be done arbitrarily; there are certain restrictions on the form of the formulae involved.

Kleene [1952a] contains a thorough analysis of the possibilities for such "permutations" of the inference rules. Since there are on the order of ten rules, there are on the order of 100 cases to consider— a number which is doubled when you realize that some permutations are permissible classically, but not intuitionistically. Each individual case is simple: you can work them out yourself in a few lines by writing out the last two lines of a derivation and then trying to reverse the order of the last two rules. Many cases are obviously permutable, for example if one rule affects only the antecedent and the other only the succedent. However, it is not the case that any two rules permute.

Remark (examples of non-permutable pairs of rules): The permutation of certain pairs is blocked by the "restriction on variables". An example of this kind of non-permutable inference is given by the proof of the sequent $\forall x A \Rightarrow \exists y A[y/x]$, which must have the $\forall \Rightarrow$ inference last. In classical logic, this is the only block to the permutation of inferences. In intuitionistic logic, there are others: for example the sequent $A \vee (B \vee C) \Rightarrow (A \vee B) \vee C$ can only be proved by opening up the left side first.

In this paper we need only certain cases of Kleene's Permutation Lemma (Lemma 7 of Kleene [1952a]). We need those cases, however, not (only) for the

system **G1** treated in Kleene [1952a] but also for the systems **G3**, **G4**, and **G**. In the interest of a complete and self-contained presentation, we therefore give the proof of the cases we need.

Lemma 2 (Permutation Lemma) (Kleene [1952a]) **G** *derivations (***G3** *derivations; strict* **G** *derivations) can be transformed so as to permute the order of applications of the rules* $\rightarrow\Rightarrow$, $\forall\Rightarrow$, *and* $\Rightarrow\exists$, *and we can permute* $\forall$*seq below* $\Rightarrow\wedge$. *(except of course in the case when the principal formula of the first inference is a subformula of the principal formula of the second one).*

This includes permuting two applications of $\rightarrow\Rightarrow$. *When two applications of the* $\rightarrow\Rightarrow$ *rule are permuted, the principal formulae of the inferences are the same before and after the transformation.*

In addition, we can permute $\forall$*seq below* $\Rightarrow\wedge$. [9]

Proof. Consider for example the permutation of the rules $\forall\Rightarrow$ and $\rightarrow\Rightarrow$. To bring $\rightarrow\Rightarrow$ 'below' $\forall\Rightarrow$ we transform the deduction steps [10]

$$\begin{array}{c}\forall xA, C\rightarrow D, \Gamma\Rightarrow Q\\ A[t/x], \forall xA, C\rightarrow D, \Gamma\Rightarrow Q\\ A[t/x], \forall xA, (C\rightarrow D), \Gamma\Rightarrow C\\ D, (C\rightarrow D), A[t/x], \forall xA, \Gamma\Rightarrow Q\end{array}$$

into the steps

$$\begin{array}{c}\forall xA, C\rightarrow D, \Gamma\Rightarrow Q\\ \forall xA, (C\rightarrow D), \Gamma\Rightarrow C\\ A[t/x], \forall xA, (C\rightarrow D), \Gamma\Rightarrow C\\ \forall xA, D, (C\rightarrow D), \Gamma\Rightarrow Q\\ A[t/x], \forall xA, D, (C\rightarrow D, \Gamma\Rightarrow Q\end{array}$$

Next consider the reverse transformation, in which we desire to bring an application of $\forall\Rightarrow$ below an application of $\rightarrow\Rightarrow$. The derivation we start with thus contains:

$$\forall xA, C\rightarrow D, \Gamma\Rightarrow Q$$

[9] The *principal formula* of an inference is the one whose connective is introduced by the inference.

[10] where the parentheses around $(C\rightarrow D)$ indicate that this formula may possibly be omitted at the indicated location, if it is obtained by substitution from a matrix of a universally quantified member of $\Gamma, \forall xA$. If we consider only **G3** derivations, no such omissions are allowed. If we consider *strict* **G** derivations, they will occur whenever allowed. In cases in which some possible omissions are not made, some inferences by $\forall\Rightarrow$ may have to be inserted in the exhibited transformed derivations, so that the same formula is either omitted or not throughout a given exhibited piece of a derivation.

$$\forall xA, D, (C \to D), \Gamma, \Rightarrow Q$$
$$A[t/x], \forall xA, D, (C \to D), \Gamma \Rightarrow Q$$
$$\forall xA, (C \to D), \Gamma \Rightarrow C$$
$$A[s/x], \forall xA, (C \to D), \Gamma \Rightarrow C$$

Note that this is not quite the reverse of the preceding situation since two instances of A are used, i.e. s may not be the same term as t. The calculus **G** does not permit simply padding the antecedent with more hypotheses as you go up the tree. However, extra hypotheses can be introduced by an extra application of $\forall \Rightarrow$, resulting in the intermediate tranformation

$$\forall xA, C \to D, \Gamma \Rightarrow Q$$
$$\forall xA, D, (C \to D), \Gamma, \Rightarrow Q$$
$$A[t/x], \forall xA, D, (C \to D), \Gamma \Rightarrow Q$$
$$A[s/x], A[t/x], \forall xA, D, (C \to D), \Gamma \Rightarrow Q$$
$$\forall xA, (C \to D), \Gamma \Rightarrow C$$
$$A[s/x], \forall xA, (C \to D), \Gamma \Rightarrow C$$
$$A[t/x], A[s/x], \forall xA, (C \to D), \Gamma \Rightarrow C$$

The extra instances of A on the 'top' two lines can be carried 'upwards' in the antecedents of preceding lines until they reach an axiom. The final transformation is then

$$\forall xA, C \to D, \Gamma \Rightarrow Q$$
$$A[t/x], \forall xA, (C \to D), \Gamma \Rightarrow Q$$
$$A[t/x], A[s/x], \forall xA, (C \to D), \Gamma \Rightarrow Q$$
$$A[s/x], A[t/x], \forall xA, D, (C \to D), \Gamma \Rightarrow Q$$
$$A[t/x], A[s/x], \forall xA, (C \to D), \Gamma \Rightarrow C$$

Next we show the permutability of rule $\to\Rightarrow$ and rule $\Rightarrow \exists$. Suppose we are given a derivation containing

$$A \to B, \Gamma \Rightarrow \exists xC$$
$$A \to B, \Gamma \Rightarrow C[t/x]$$
$$(A \to B), \Gamma \Rightarrow A$$
$$B, (A \to B), \Gamma \Rightarrow C[t/x]$$

It can be transformed to

$$A \to B, \Gamma \Rightarrow \exists xC$$

$$(A \to B), \Gamma \Rightarrow A$$
$$B, (A \to B), \Gamma \Rightarrow \exists x C$$
$$B, (A \to B), \Gamma \Rightarrow C[t/x]$$

and this transformation is reversible, so that rules $\Rightarrow \exists$ and $\to\Rightarrow$ commute.

Rules $\Rightarrow \exists$ and $\forall \Rightarrow$ affect opposite sides of the $\Rightarrow$ sign, and hence obviously commute.

It remains to permute two different applications of $\to\Rightarrow$. There are two cases, according as the 'higher' application is on the right branch or the left branch of the 'lower' application. We take the right branch case first. Suppose given:

$$A \to B, C \to D, \Gamma \Rightarrow Q$$
$$(A \to B), C \to D, \Gamma \Rightarrow A\ *$$
$$B, (A \to B), C \to D, \Gamma \Rightarrow Q$$
$$B, (A \to B), (C \to D), \Gamma \Rightarrow C$$
$$D, B, (A \to B), (C \to D), \Gamma \Rightarrow Q$$

We transform this to

$$A \to B, C \to D, \Gamma \Rightarrow Q$$
$$A \to B, (C \to D), \Gamma \Rightarrow C$$
$$(A \to B), (C \to D), \Gamma \Rightarrow A$$
$$B, (A \to B), (C \to D), \Gamma \Rightarrow C$$
$$D, A \to B, (C \to D), \Gamma \Rightarrow Q$$
$$D, (A \to B), (C \to D), \Gamma \Rightarrow A\ *$$
$$B, D, (A \to B), (C \to D), \Gamma \Rightarrow Q$$

Of the four 'top' nodes in this (partial) derivation, three are 'top' nodes in the given derivation (so we can graft on the subtrees at those nodes). The fourth node, tagged $(*)$, is like the tagged node in the given derivation, except it has the extra formula D in the antecedent. We can graft on the subtree 'above' the tagged node in the given derivation, provided we write D in the antecedent of every formula in that subtree. (This will not affect the validity of any inference, including the axioms at leaves, since D is already a subformula of the antecedent, so its free variables cannot cause any violation of the restriction on variables.)

This transformation, unlike the others we have constructed, may increase the number of applications of $\to\Rightarrow$, as the implication which is moved 'upwards' has to be opened up twice, and a subtree of the old derivation has to be duplicated in the new.

Finally we consider the case of permuting an application of $\rightarrow\Rightarrow$ with another application on the left branch of the first. Suppose given:

$$\begin{gathered} A \rightarrow B, C \rightarrow D, \Gamma \Rightarrow Q \\ (A \rightarrow B), C \rightarrow D, \Gamma \Rightarrow A \\ (A \rightarrow B), (C \rightarrow D), \Gamma \Rightarrow C \\ D, (A \rightarrow B), (C \rightarrow D), \Gamma \Rightarrow A \\ B, (A \rightarrow B), C \rightarrow D, \Gamma \Rightarrow Q \end{gathered}$$

We transform this to

$$\begin{gathered} A \rightarrow B, C \rightarrow D, \Gamma \Rightarrow Q \\ A \rightarrow B, (C \rightarrow D), \Gamma \Rightarrow C \\ D, A \rightarrow B, (C \rightarrow D), \Gamma \Rightarrow Q \\ D, (A \rightarrow B), (C \rightarrow D), \Gamma \Rightarrow A \\ B, D, (A \rightarrow B), (C \rightarrow D), \Gamma \Rightarrow Q \end{gathered}$$

whose 'top' nodes are the same as those of the given derivation.

Now there is only one more permutation to consider: permuting $\forall \Rightarrow$ below $\Rightarrow \wedge$. Suppose given a derivation ending

$$\begin{gathered} \forall x P, \Gamma \Rightarrow A \wedge B \\ \forall x P, \Gamma \Rightarrow A \\ P[t/x], \forall x P, \Gamma \Rightarrow A \\ \forall x P, \Gamma \Rightarrow B^* \end{gathered}$$

We transform this to

$$\begin{gathered} \forall x P, \Gamma \Rightarrow A \wedge B \\ P[t/x], \forall x P, \Gamma \Rightarrow A \wedge B \\ P[t/x], \forall x P, \Gamma \Rightarrow A \\ P[t/x], \forall x P, \Gamma \Rightarrow B^* \end{gathered}$$

Note that the starred line isn't exactly the same; there is an extra formula $P[t/x]$ in the succedent of the transformed derivation. We will have to copy this formula to the succedent of every sequent 'above' the starred line (that is, nearer the axioms) in the original derivation. In so doing, we won't violate any conditions on variables, since the condition on variables is satisfied in the original derivation. That completes the proof.

3 Explaining Prolog by Gentzen Sequents

We view Prolog's deduction algorithm as the construction of a formal derivation in an intuitionistic sequent calculus. The Prolog program, a finite list of clauses, constitutes the antecedent of a sequent, and the query constitutes the succedent.

The clauses in the program go in the antecedent as implications, while the goal is existentially quantified in the succedent: [11]

$$(\forall x)(Hyp_1 -> P_1), (\forall y)(Hyp_2 -> P_2), \ldots => (\exists z)Goal$$

Note that since "the scope of a Prolog variable is the clause in which it occurs", the clauses are universally quantified in the antecedent. Here the quantifier $\exists z$ is short for a finite sequence of quantifiers over every variable free in *Goal*, and similarly for the universal quantifiers on the left.

To make matters as clear as possible: the above sequent is the expression as a Gentzen sequent of a Prolog program (given by the antecedent) together with a query. Normally a user of Prolog would enter the clauses in the form P_n :- Hyp_n, and Hyp_n would be a conjunction of atomic formulae. The system understands the universal quantifiers implicitly. These clauses would be put in a program file and consulted, after which the query *Goal* would be typed to the Prolog interpreter prompt. The existential quantifier(s) are also understood implicitly by the Prolog system. When we refer to "clauses" in this paper, we will be speaking of implications in the antecedent of a sequent; the usual form of clauses in connection with resolution will not be mentioned. (We do count a clause with no "body", i.e. a universally quantified atomic formula in the antecedent counts as a "clause".)

The following definitions make the above considerations precise:

Definition 1 *A "Prolog clause" is a formula of one of the forms* $\forall x(Con)$ *or* $\forall x(Hyp \rightarrow Con)$*, where Con is atomic, and Hyp is either atomic or a conjunction of (possibly many) atomic formulae. The universal quantifier need not actually occur.*

Definition 2 *A Prolog sequent is a sequent whose succedent is either atomic, a conjunction of atomic formulae, or an existentially quantified atomic formula, and whose antecedent contains (only) Prolog clauses as just defined.*

A *closed* Prolog sequent corresponds to a Prolog program and query. Prolog sequents with free variables may arise in the course of Prolog's computations, as we shall see below.

Prolog's deduction algorithm tries to unify *Goal* with the "head" P_n of one of the clauses in the antecedent. Let us suppose for notational simplicity that *Goal* unifies with P_1. The deduction step which unifies *Goal* with $P1$ involves

[11] This is one important place where our treatment differs from Bowen [1980]. He puts the program clauses as separate sequents, and considers them as "sequent axioms", thus treating Prolog deductions as applications of the cut rule instead of constructions of cut-free derivations.

the most general unifier θ of $Goal$ and P_1. Suppose for the moment that we are in the simplest case, in which $Goal\theta$ has no free variables. Prolog's computation in this case is represented by the following incomplete **G** proof

$$\forall x(Hyp_1 \to P_1), \forall y(Hyp_2 \to P_2), \ldots => \exists z Goal$$
$$\forall x(Hyp_1 \to P_1), \forall y(Hyp_2 \to P_2), \ldots => Goal\theta$$
$$Hyp_1\theta \to P_1\theta, \forall x(Hyp_1 \to P_1), (\forall y)(Hyp_2 \to P_2), \ldots => Goal\theta$$
$$\forall x(Hyp_1 \to P_1), \ldots \Rightarrow Hyp_1\theta$$
$$P_1\theta, \forall x(Hyp_1 \to P_1), \ldots \Rightarrow Goal\theta$$

The last sequent is an axiom, since $P_1\theta$ is the same formula as $Goal\theta$ (that's what it means that θ unifies $Goal$ and P_1), so that formula occurs on both sides of $\Rightarrow$. The computation will continue in an attempt to construct a proof of the previous sequent. The last inference is by rule $\to\Rightarrow$; compare it with the list of rules to see why.

Note that we have allowed conjunction in the succedent; the main reason for doing so is that such formulae will arise in derivations anyway, when a "clause" containing a conjunction is "opened up". It is customary to extend Prolog to allow disjunction in the query and the heads of the clauses, but these are easily reduced to formulae of the above form by the introduction of new predicate letters. Alternately, one can easily extend the description of Prolog's algorithm in terms of Gentzen sequents to this somewhat larger fragment. We shall not take the space to do so, since we shall extend the algorithm to all of first-order predicate calculus later in the paper.

Non-deterministic versus deterministic Prolog. Implementations of Prolog search for a proof of the goal in a deterministic manner. By a *non-deterministic Prolog deduction* we mean a deduction as described above in which the choice of clause $Hyp_n \to P_n$ can be made at will (non-deterministically). In practice, an implementation of Prolog chooses the leftmost clause whose head will unify with the goal; and it is possible that the search regresses infinitely along this branch, while another choice would have succeeded in producing a proof. All known metamathematical results on the semantics of Prolog refer to non-deterministic proofs, as will the results of this paper. Nevertheless, our implementation of the algorithm described here, like Prolog, makes a deterministic choice. Experiments show that, like Prolog, it still works.

Note that in the case of Horn clause syntax, the succedent is always atomic or atomic preceded by $\exists$; so it can unify only with conclusions of the clauses, i.e. with formulae P_n, where $\forall x_n(Hyp_n \to Con_n)$ is one of the "clauses" of the Prolog program represented by the antecedent, or possibly with formulae generated from some P_n by applying some other substitution (lower down the proof tree). These formulae P_n are positive in the antecedent, so actual Prolog deductions fit the formal definition just given. Moreover, restricted to Horn clause syntax, Prolog deductions as just defined clearly correspond to (non-deterministic) Prolog deductions as informally defined above.

More generally, if $Goal\theta$ does still contain free variables, the Prolog computation algorithm constructs a cut-free ($\mathbf{G}$) proof, but it does not do it line-by-line. Instead, the terms needed in the hypotheses of the $\forall \Rightarrow$ and $\Rightarrow \exists$ rules are constructed gradually, only being settled on after the computation has progressed some lines above the application of the rule in question.

Prolog explained by Gentzen proofs, using Prolog for the explanation

We give here the fragment of our theorem-prover which is needed for the derivation of Prolog sequents. This Prolog program defines the predicate `derive`($\Gamma \Rightarrow A$) which succeeds if the system can construct a Gentzen proof (in $\mathbf{G}$) of $\Gamma \Rightarrow A$. We write $\forall x A$ in Prolog as `all`(x, A). The predicate `prove`(A) is defined to mean, "find a derivation of the sequent $\Rightarrow A$ (with empty antecedent)". Sequents are represented in Prolog using the infix operator $\Rightarrow$, where the antecedent is a list of formulae and the succedent a single formula (with `false` representing the empty succedent). Logical variables ("object variables") are represented as Prolog atoms. Prolog variables (beginning with upper-case letters) range over object terms.

The Prolog Program

```
% construct a strict G derivation of a Prolog sequent
 axiom((Gamma ⇒ A)):- member(A,Gamma).
 derive(( Gamma => T=T )).
        %this accounts for Prolog's treatment of equality
 derive(( Gamma => C )):-                    %rule -> =>
    memberandrest( (A->B), Gamma,Delta),
       %Delta is what's in Gamma besides (A->B)
    A  \== C,    %prevent an obviously redundant proof
    axiom(( [B|Gamma] => C )),
    derive((Delta => A)).
 derive(( Gamma => and(A,B)):-               %rule => &
    derive(( Gamma => A )),
    derive(( Gamma => B )).
 derive((Gamma => B)) :-   %rule all =>  (list quantifier)
   member(all([X|Rest],A),Gamma),
   findall(_,member(_,[X|Rest]),NewVarList),
   list_fsubst(NewVarList,[X|Rest],A,NewA),
   not fmember(NewA,Gamma),
   derive(( [NewA|Gamma] => B )).
 derive((Gamma => B)) :-   %rule all =>  (single quantifier)
   member(all(X,A),Gamma),
   fsubst(_,X,A,NewA),
   not fmember2(NewA,Gamma),
         %don't make multiple instances of same clause
   derive(( [NewA|Gamma] => B )).
```

The predicates `fmember`, `fmember2`, `fsubst`, and `memberandrest` are not printed here, for lack of space. The program can still be understood by means of the following descriptions of these predicates:

`fmember(A,L)` generates members `A` of a given list `L`, or tests whether `A` belongs to `L`, but without unification: thus `A` and `L` can contain Prolog variables which will be treated like atoms. `fmember2` is similar, but it does not distinguish terms that differ only be renaming of Prolog variables. Similarly, `fsubst(New,Old,Term,Ans)` substitutes `New` for `Old` in `Term` obtaining `Ans`, but treating Prolog variables as atoms (not subject to unification). They are distinguished from the more ordinary `member` and `subst` by the prefix `f` for "free variables". `list_subst` takes a list of items `New` to be substituted respectively for the members of a list of items `Old` and uses `fsubst` to carry out the substitutitions. The line using `findall` in the rule for a list quantifier just generates a list of new Prolog variables of the right length. The predicate `memberandrest(Mem,List,Rest)` generates members `Mem` of `List`, instantiating `Rest` to `List` with (the occurrence in question of) `Mem` deleted.

The main theorem explaining Prolog in terms of Gentzen sequents can now be stated:

Theorem 1 *(1) Let* $\Gamma \Rightarrow \exists x A$ *be a Prolog sequent representing a Prolog program* Γ *and goal* A*. Then the goal* A *succeeds with program* Γ *if and only if the goal* `derive`$(\Gamma \Rightarrow \exists x A)$ *succeeds with the program listed above.*

(2)If we give the program the query `derive`$(\Gamma \Rightarrow A[X/x])$*, in which the succedent contains Prolog variables (which range over terms), then the Prolog interpreter will supply an instantiating term such that* $\Gamma \Rightarrow A[t/x]$ *is provable in* **G**.

This is two theorems: one when "succeed" is interpreted non-deterministically, and another when it has the deterministic interpretation given by an actual interpreter.

Remark: While `derive` is obviously sound, it is not obviously complete, since it requires the right branch of an application of rule $\rightarrow\Rightarrow$ to be an axiom already. The issue of completeness is addressed in the next section.

Proof. (1) follows from (2). We prove (2) by induction on the length of Prolog computations from the given program. Suppose then that $\Gamma \Rightarrow A$ is a closed Prolog sequent, and suppose that Prolog's computation of the query A relative to program Γ begins by unifying A with the head P of a member $\forall x(H \rightarrow P)$ of Γ. For simplicity let us assume that the members of Γ have at most one universal quantifier. Evidently the program above will begin by applying the clause of `derive` corresponding to rule $\forall \Rightarrow$ with this same member of Γ as principal formula of the inference. That will result in an attempt to `derive` the sequent $H\theta \rightarrow P\theta, \Gamma \Rightarrow A\theta$, where θ is the most general unifier of A and P. Then rule $\rightarrow\Rightarrow$ will be applied, resulting in an attempt to `derive` $\Gamma \Rightarrow H\theta$. Note that the formula $H\theta \rightarrow P\theta$ has been dropped from the antecedent in accordance with the rules of **G**.

If H is a conjunction, an application of the clause labelled "rule => &" will reduce to the case H is atomic, so that $\Gamma \Rightarrow H\theta$ is again a Prolog sequent. Now the Prolog computation proceeds as specified by the Prolog sequent $\Gamma \Rightarrow H\theta$. By hypothesis, this Prolog computation succeeds; so by induction hypothesis, the attempt to `derive` this sequent also succeeds. But then the original call to `derive` succeeds. That completes the proof.

Explanation of Prolog by Gentzen proofs, not using Prolog

The remainder of this section serves two purposes: it introduces the technical tool of "extended" Gentzen sequents and derivations, and it uses that tool to explain Prolog in terms of Gentzen sequents, in a way not dependent on a prior understanding of Prolog.

To see how Prolog works in terms of Gentzen sequents, let us consider an example.

Consider the Prolog sequent (program and goal) given by

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \exists z \gamma(z) \tag{1}$$

Prolog tries to construct a proof whose last inference is by the $\Rightarrow \exists$ rule; for this it needs a term t so that $\gamma[t/z]$ can be put in the succedent. For reasons that will be clear in a minute, we prefer to place the emphasis on the substitution η such that $t = \eta x$. At first we cannot determine t, we can only place a constraint upon η: by unifying $\gamma(z)$ with the head of one of the clauses in the succedent, we can see that ηx should have the form $(x\theta)^3$, where θ is a substitution to be determined.

We then consider the sequent

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((x\theta)^3) \tag{2}$$

and open up the $\forall$ on the left with the $\forall \Rightarrow$ rule (working in what would traditionally be the upwards direction of a proof, i.e. towards the axioms):

$$\beta(x\theta) \to \gamma((x\theta)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((x\theta)^3) \tag{3}$$

Next we use the $\to\Rightarrow$ rule (in reverse) to open up the first formula. The two sequents which occur above the line of this inference are:

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(x\theta) \tag{4}$$

and

$$\gamma((x\theta)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((x\theta)^3) \tag{5}$$

The last sequent (5) is an axiom; not by accident, as that was how we chose the term $t = (x\theta)^3$ in the first place.

The sequent (4) is analyzed as follows: Prolog tries to unify the succedent formula with the head of one of the clauses. The only possibility in this case is to unify $\beta(u^2)$ with $\beta(x\theta)$. Hence we impose a constraint on θ: it must

satisfy $x\theta = (u^2)\delta$ for some substitution δ. (If the variable u had occurred in the succedent, we should have renamed it first.) Introducing a symbol for this unknown substitution δ, we work upwards in the proof using the rule $\forall \Rightarrow$:

$$\alpha(u\delta) \to \beta((u\delta)^2), \beta(x\theta) \to \gamma((x\theta)^3), \alpha(2),$$
$$\forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(x\theta) \qquad ()6)$$

Now we can use the rule $\to\Rightarrow$ as before, generating the following two sequents:

$$\beta(x\theta) \to \gamma((x\theta)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \alpha(u\delta) \quad (7)$$

$$\beta((u\delta)^2), \beta(x\theta) \to \gamma((x\theta)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(x\theta) \quad (8)$$

As before, the last sequent (8) is an axiom by our choice of (constraint upon) δ. We must continue from the sequent (7), by unifying the succedent with the head of a clause in the antecedent. The only possibility is the clause $\alpha(2)$. If $\alpha(u\delta)$ is to unify with this, we must have $u\delta = 2$. This determines $x\theta = (u\delta)^2 = 2^2$, and hence $\eta x = (x\theta)^3 = (2^2)^3$. Substituting these values for the symbols η,θ,and δ, we find that a **G** proof has been constructed:

Example 1.

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \exists z\gamma(z)$$
$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((2^2)^3)$$
$$\beta(2^2) \to \gamma((2^2)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((2^2)^3)$$
$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(2^2)$$
$$\gamma((2^2)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma((2^2)^3)$$
$$\alpha(2) \to \beta(2^2), \beta(2^2) \to \gamma((2^2)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(2^2)$$
$$\beta(2^2) \to \gamma((2^2)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \alpha(2)$$
$$\beta(2^2), \beta(2^2) \to \gamma((2^2)^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(2^2)$$

This example shows how the Prolog deduction algorithm can be regarded as constructing (or attempting to construct) a **G** proof of a sequent representing the program and query. The algorithm itself can be precisely specified as a systematic attempt to proceed from the given sequent to axioms by applying the rules of inference, leaving symbols for unknown substitutions to be determined, and simultaneously building up a system of equational constraints on these symbols for substitutions. The algorithm terminates successfully if axioms are reached. When that happens, the system of equational constraints is certainly solvable, since its form is just a chain of substitutions.

We next define a more precise notation for this process. Instead of permitting symbols for substitutions in the derivation itself, we write the symbols for substitutions outside the formula on each line,

$$\Gamma \Rightarrow A \quad : \theta$$

This notation will be precisely defined. In order to explain Prolog in terms of Gentzen sequents, which is the purpose of this section, we need only the first four clauses of the following definition; the rest are needed only to generalize this explanation to our theorem-prover (GENTZEN) for the entire intuitionistic predicate calculus.

We shall need an alphabet of variables distinct from the "object variables" of our logical language. The object variables have been defined as Prolog atoms (other than `true` and `false`), i.e. they all begin with lower-case letters. By a "Prolog variable" we mean a syntactic object specified by the same syntax as Prolog usually accepts for variables, i.e. a word beginning with an upper-case letter.[12]

Definition 3 *(Extended Formulae). These are just ordinary formulae in which Prolog variables are allowed to occur free.*

Definition 4 *(Extended Sequents). These are expressions of the form* $\Gamma \Rightarrow A \quad : \theta$*, where* Γ *is a finite set of extended formulae,* A *is an extended formula, and* θ *is a substitution. The substitution* θ *must act non-trivially only on Prolog variables.*

Note that the substitutions occurring in extended sequents will in general act non-trivially on variables which do not occur in the formula part of the extended sequent. Expressions of the form $\Gamma \Rightarrow A \quad : \theta$ will always stand for extended sequents.

Definition 5 *(Extended* **G** *Derivations). The following clauses define the notion*
$\vdash \Gamma \Rightarrow A \quad : \theta$

Indentation is used for 'if' just as in specifying rules of inference. Certain premises of rules $\rightarrow\Rightarrow$, $\vee \Rightarrow$, and $\wedge \Rightarrow$ are parenthesized, with a similar meaning as in the specification of the rules of **G** . Namely, the premise $A \rightarrow B$ can be omitted if $A\theta \rightarrow B\theta$ is a substitution instance of the matrix of a universally quantified formula in $\Gamma\theta$.

$$B, \Gamma \Rightarrow A \quad : \theta \quad \textit{if } B\theta = A\theta \qquad (\texttt{axiom})$$

$$\begin{array}{c} A \rightarrow B, \Gamma \Rightarrow C \quad : \theta \qquad (\rightarrow\Rightarrow) \\ (A \rightarrow B), \Gamma \Rightarrow A \quad : \theta \\ B, (A \rightarrow B), \Gamma \Rightarrow C \quad : \theta \end{array}$$

[12] Calling them "Prolog variables" is suggestive of the implementation of `derive` used in this paper, where actual Prolog variables are used for "Prolog variables". However, this is not the case in some other implementations; the only point is to distinguish a kind of variable distinct from the object variables of the language.

$$\frac{\Gamma \Rightarrow \exists x A \quad : \theta}{\Gamma \Rightarrow A[X/x] \quad : \theta \text{ (where } X \text{ is a Prolog variable)}} \; (\Rightarrow \exists)$$

$$\frac{\forall x A, \Gamma \Rightarrow C \quad \theta}{A[X/x], \Gamma \Rightarrow C \quad : \theta \text{ (where } X \text{ is a Prolog variable)}} \; (\forall \Rightarrow)$$

$$\frac{\Gamma \Rightarrow \forall x A \quad : \theta}{\Gamma \Rightarrow A \quad : \theta \ (x\theta \text{ not free in } \Gamma\theta)} \; (\Rightarrow \forall)$$

$$\frac{\exists x A, \Gamma \Rightarrow B \quad : \theta}{A, \Gamma \Rightarrow B \quad : \theta \ (x\theta \text{ not free in } \Gamma\theta)} \; (\exists \Rightarrow)$$

The remaining clauses of the definition are exactly the same as the propositional rules of inference, but with $: \theta$ written beside each formula.

Example 2: The extended derivation corresponding to Example 1 above is as follows: (When the page is not wide enough to place the annotations on the same line as the formulae, they are placed on the next line.)

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \exists z \gamma(z) \quad : Z = (2^2)^3, X = 2^2, U = 2$$

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma(Z) \quad : Z = (2^2)^3, X = 2^2, U = 2$$

$$\beta(X) \to \gamma(X^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma(Z)$$
$$: Z = (2^2)^3, X = 2^2, U = 2$$

$$\alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(X) \quad : Z = (2^2)^3, X = 2^2, U = 2$$

$$\gamma(X^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \gamma(Z)$$
$$: Z = (2^2)^3, X = 2^2, U = 2$$

$$\alpha(U) \to \beta(U^2), \beta(X) \to \gamma(X^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(X)$$
$$: Z = (2^2)^3, X = 2^2, U = 2$$

$$\beta(X) \to \gamma(X^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \alpha(U)$$
$$: Z = (2^2)^3, X = 2^2, U = 2$$

$$\beta(U^2), \beta(X) \to \gamma(X^3), \alpha(2), \forall u(\alpha(u) \to \beta(u^2)), \forall x(\beta(x) \to \gamma(x^3)) \Rightarrow \beta(x)$$
$$: Z = (2^2)^3, X = 2^2, U = 2$$

Note that an extended derivation gives rise to an actual **G** derivation if we carry out the substitutions indicated on each line.

Lemma 3 *If* $\vdash \Gamma \Rightarrow C \quad : \theta$, *then there is a* **G** *derivation of* $\Gamma\theta \Rightarrow C\theta$.

Proof. Induction on the length of the extended derivation of $\Gamma \Rightarrow C \quad : \theta$. That completes the proof.

We are now in a position to explain in terms of Gentzen sequents what Prolog "really" does:

Definition 6 *An extended derivation is called a "Prolog derivation" if it satisfies the following conditions at each application of rule* $\rightarrow\Rightarrow$, *and if* θ *cannot be replaced in the derivation by any more general substitution:*

$$\frac{A \rightarrow B, \Gamma \Rightarrow C \quad : \theta}{A \rightarrow B, \Gamma \Rightarrow A \quad : \theta} \quad (\rightarrow\Rightarrow)$$
$$B, A \rightarrow B, \Gamma \Rightarrow C \quad : \theta$$

Then θ is required to unify C with B. That is, the right premise of the inference is required to be a logical axiom.

The following "Observation" codifies the sense in which "what Prolog does is compute extended **G** derivations":

Observation 1 *A Prolog computation with program* Γ *and query* $A[X/x]$ *constructs a substitution* θ *and a Prolog* **G** *derivation of* $\Gamma \Rightarrow A[X/x] \quad : \theta$.

One who has understood the definitions and who knows Prolog will recognize the truth of this "Observation". It is difficult to "prove" it, since it depends on a precise definition of "Prolog computation". We could take the definition in Lloyd [1984] in terms of SLD resolution, in which case the Observation would be a Theorem, but we feel that extended **G** derivations are closer to what Prolog interpreters actually do than SLD resolution. We therefore omit any formal definition and corresponding proof. The reader who is not familiar with Prolog may take the "Observation" simply as an "Explanation": this is what Prolog does.

For a theorem with essentially the same content, we refer to Theorem 1 above.

In this section, we are interested only in derivations of Prolog sequents. In that case, the antecedent contains only formulas $\forall x(Hyp \rightarrow Con)$ or $\forall x(Con)$, where Con is atomic, and the succedent is always atomic or an existentially quantified atomic formula, so the above condition amounts to requiring that the goal unify with the head of a clause in the program. Thus the unification steps allowed by the above definition are just those permitted by Prolog. In other words, we have the converse of Observation 1:

Observation 2 *Any Prolog derivation of an extended Prolog sequent* $\Gamma \Rightarrow Goal \quad : \theta$ *corresponds to a (non-deterministic) Prolog computation proving* $Goal$, *with answer substitution* θ, *from the program* Γ.

Like Observation 1, this theorem could be "proved" if we accept some other concept as the definition of "Prolog computation", but we prefer to let it stand as an intuitively clear characterization of the notion, as we feel it is closer to what Prolog computation systems actually do than the notion of SLD resolution.

We summarize, somewhat less formally, the main result:

Prolog computations correspond to the construction of cut-free extended derivations from the root up, in which process, at each application of rule $\rightarrow\Rightarrow$, the right premise is already an axiom on the basis of the substitution constructed so far.

4 Completeness of Prolog

The completeness of Prolog deductions for the Horn clause fragment of logic is the fundamental result on the semantics of Prolog. See Lloyd [1981] (Chapters 1 and 2) for the standard proof, in terms of Herbrand models and resolution. Here we give an alternate proof, which is informative because it shows that the fundamental result on semantics of Prolog depends on proof-theoretical properties of Gentzen calculi.

At first, the author thought that the completeness of Prolog would boil down to just the cut-elimination theorem in the Horn-clause fragment. This was not quite the case: Prolog deductions have the important restriction that the right premise of an inference by rule $\rightarrow\Rightarrow$ must be an axiom. We must not only rely on cut-elimination, but also on the Permutation Lemma to show that proofs satisfying this restriction are complete.

The reader should now glance at the statement of the theorem near the end of this section, before continuing in logical fashion to read the lemmas leading up to it.

We begin with a lemma that shows it doesn't matter whether we use classical or intuitionistic logic in Horn clause deductions:

Lemma 4 *A Prolog sequent has a proof in intuitionistic* **G1** *if and only if it has a proof in classical* **G1**.

Proof. Suppose we have a Prolog program Γ (regarded as a set of **G** formulae) and suppose the sequent $\Gamma \Rightarrow \exists x A$ is provable in classical **G1**. Here A is atomic and we assume $\exists x A$ possibly abbreviates several existential quantifiers; but for notational simplicity we shall write only one.

The only propositional connectives in the sequent are $\rightarrow$, which occurs only in the antecedent and is not iterated, and , which occurs only in the left part of implications in the antecedent, and possibly in the succudent. It follows that the **G1** proof in question uses only quantifier rules, the rule $\rightarrow\Rightarrow$, and the rule $\Rightarrow$. None of these rules introduces two formulae on the right as we go up the tree: that is done only by the classical rule

$$\frac{\neg A, \Gamma \Rightarrow \Theta, A}{\neg A, \Gamma \Rightarrow \Theta}$$

Consequently the proof is actually in intuitionistic **G3**. (Remember the only difference between intuitionistic and classical sequent calculi is the restriction to one formula in the succedent in the intuitionistic calculi.) [13] That completes the proof.

The essence of the problem is now clear:

(1) An arbitrary **G** proof can introduce unknown substitutions at rules $\forall \Rightarrow$ and $\Rightarrow \exists$. Prolog, on the other hand, admits only substitutions produced by certain specified unifications. Are these specifications general enough to produce a proof of every sequent that has some **G** proof?

(2) An arbitrary **G** proof can use both branches of applications of rule $\rightarrow\Rightarrow$, while a Prolog derivation can use only the left branch. Can the use of the right branch be avoided?

The answer to both questions is yes. To some extent they can be answered separately; we first address (2).

Lemma 5 *Let $\Gamma \Rightarrow Goal$ be a Prolog sequent derivable in* **G**. *Then it has a* **G***-derivation in which at every application of rule $\rightarrow\Rightarrow$, the right branch is an axiom.*

Proof. First, by Gentzen's cut-elimination theorem, we may find a cut-free derivation of the given sequent. In this derivation, the existential quantifier in the succedent (if any) must be introduced after all applications of rule $\rightarrow\Rightarrow$, since no existential quantifiers occur in the antecedent. We may use the Permutation Lemma to bring applications of $\Rightarrow \exists$ to the root of the tree. Then, again by the Permutation Lemma, we may bring all applications of $\forall \Rightarrow$ towards the root of the tree until there are no applications of other rules except $\Rightarrow \exists$ below them. At some points in the derivation, as we move "up" the tree (towards the axioms), an application of $\rightarrow\Rightarrow$ may create a conjunction in the succedent. If this happens, we may assume (by the Permutation Lemma) that the conjunction is immediately broken up by (enough applications of) rule $\Rightarrow \wedge$ until the succedents are atomic.

We now give an algorithm based on the Permutation Lemma for transforming the given derivation from the form explained so far to a derivation of the desired form.

Case 1: If the last inference is by rule $\Rightarrow \exists$, $\forall \Rightarrow$, or $\Rightarrow \wedge$, just apply the algorithm to transform the derivations of the premise(s) of the last inference.

Case 2: If the last inference is an application of rule $\rightarrow\Rightarrow$, then by the form required of the input, the only rules used in the derivation are $\rightarrow\Rightarrow$ and $\Rightarrow \wedge$, and $\Rightarrow \wedge$ is used only to break up conjunctions in the premises of an application of $\rightarrow\Rightarrow$. Let the sequent derived have atomic succudent D. Trace upwards (towards the axioms), taking the right branch at all applications of $\rightarrow\Rightarrow$, so that D is always the succedent, and the rule of inference is always $\rightarrow\Rightarrow$, until we reach an axiom $D, \Delta \Rightarrow D$ at the right branch of a certain inference by $\rightarrow\Rightarrow$.

[13] Note that Gödel's double-negation interpretation is not sufficient to establish the connection between classical and intuitionistic proofs here: it would leave double negations on the prime formulae.

Then use the Permutation Lemma (that is, the algorithm implicit in its proof) to permute that inference with all the others we have traced through to find it, until it becomes the last inference. We then have a derivation of the form

$$\begin{array}{c} A \to D, \Delta \Rightarrow D \\ (A \to D), \Delta \Rightarrow A \\ D, \Delta \Rightarrow D \end{array}$$

Apply the algorithm recursively to the sequent $(A \to D), \Delta \Rightarrow A$ occurring on the left branch. The resulting derivation is tacked on to the derivation fragment shown, producing the output derivation.

This defines an algorithm: but does it converge? Yes, because applications of the permutation lemma to permute two $\to\Rightarrow$ inferences do not increase the depth of the proof tree (though they may increase the total number of lines in the proof). It follows inductively that the algorithm being defined here does not increase the depth of the proof tree. Since the recursive calls are made on shorter trees, they terminate by induction hypothesis. Hence the algorithm terminates.

It still remains to prove that the transformed derivation is a Prolog derivation. We prove this by induction as follows: If the input is an axiom, so is the output. If the last inference is by $\Rightarrow \exists$, $\Rightarrow \wedge$, then the transformed derivation has the same last rule, and the subtrees are all Prolog derivations by induction hypothesis. If the last inference is by $\to\Rightarrow$, then the last inference of the output derivation is a right-branch-an-axiom application of $\to\Rightarrow$, by the above construction, and all the subtrees are Prolog derivations by induction hypothesis. That completes the proof.

Lemma 6 (Main Lemma) *Let $\Gamma \Rightarrow Goal$ be a Prolog sequent derivable in* **G**. *Then there is a substitution θ and a Prolog derivation of $\Gamma \Rightarrow Goal \quad : \theta$.*

Proof. By the preceding lemma, there is a **G** derivation of $\Gamma \Rightarrow Goal$ in which at every application of rule $\to\Rightarrow$, the right branch is an axiom. We show how to convert such a derivation into an equivalent extended derivation. We do not proceed from the axioms to the root, since we wouldn't have any idea what substitution to annotate the axioms with. Instead, we must start from the root sequent, and proceed 'up' the tree. As we do this, we shall annotate each line with a substitution. When we pass a quantifier rule $\forall \Rightarrow$ or $\Rightarrow \exists$, so that $A[t/x]$ occurs where $\exists x A$ or $\forall x A$ did one line before, we replace $A[t/x]$ by $A[X/x]$, where X is a new Prolog variable, and annotate with $\theta, X = t$, where θ is the annotation on the conclusion of this inference. At the same time we copy the equation $X = t$ to the annotation space of all nodes we have already passed along that branch from the root. [14] When we pass an inference by $\to\Rightarrow$, we simply copy the annotation θ already given to the conclusion to both

[14] Thus the answer substitution returned will be defined on some Prolog variables that are not in the original query. This feature of GENTZEN generalizes Prolog; in a normal Prolog

premises (restricting the domain if necessary– one branch may not mention all the variables in the domain of θ, so on that branch we should not put θ but rather θ restricted to the variables occurring in that premise.) Continuing in this way, we will eventually reach the axioms. The result is an extended derivation. That completes the proof.

Theorem 2 (Prolog Completeness Theorem) *Prolog deductions are complete for the Horn-clause fragment of (classical) predicate logic. That is, every valid Prolog sequent* $\Gamma \Rightarrow Goal$ *has a Prolog deduction. Still more precisely: for some substitution* θ*, there is a Prolog deduction of* $\Gamma \Rightarrow Goal \quad : \theta$.

Proof. Suppose $\Gamma \Rightarrow \exists x A$ is a valid Prolog sequent. By the completeness of Gentzen's cut-free (classical) system **G1**, $\Gamma \Rightarrow \exists x A$ has a derivation in classical **G1**. [15] By the Lemma, it has an intuitionistic **G1** proof. By the equivalence of **G** and **G1**, it has a **G** proof. By the Main Lemma, it has a Prolog deduction. That completes the proof.

Define $\Gamma \Rightarrow Goal \quad : \theta$ to be valid if $\Gamma\theta \Rightarrow Goal\theta$ is valid. (For this to make sense, θ must eliminate all Prolog variables.) One can strengthen the statement of the completeness theorem, as suggested to me by P. Schroeder-Heister:

Theorem 3 (Extended Completeness Theorem) *Suppose* $\Gamma \Rightarrow Goal \quad : \theta$ *is valid. Then there is a substitution* ψ *agreeing with* θ *on Prolog variables occurring in* $\Gamma \Rightarrow Goal$ *and a Prolog derivation of* $\Gamma \Rightarrow Goal \quad : \psi$.

Proof. By the previous theorem there is a Prolog derivation of $\Gamma\theta \Rightarrow Goal\theta \quad : \mu$ for some substitution μ. Taking ψ to be the composition of θ and μ, we claim that there is a Prolog derivation of $\Gamma \Rightarrow Goal \quad : \psi$. This "lifting lemma" about Prolog derivations can be proved by induction on the length of the Prolog derivation of $\Gamma\theta \Rightarrow Goal\theta$. Consider the induction step: Given a derivation with last step

$$\frac{A\mu \to B\mu, \Gamma\mu \Rightarrow C\mu \quad : \theta}{A\mu \to B\mu, \Gamma \Rightarrow A\mu \quad : \theta} \; (\to\Rightarrow)$$
$$B\mu, A\mu \to B\mu, \Gamma \Rightarrow C\mu \quad : \theta$$

query we really use Prolog variables, not existentially bound object variables, and of course explicit $\forall$ never occurs. The mechanism we have defined in the text does not provide an explicit bookkeeping mechanism to explain the meaning of the "extra" variables on which the answer substitution is defined; you will have to examine the extended derivation produced.

[15] The completeness of **G1** can be proved directly as is done for a similar system in Schutte [1977] (p. 28), or derived from Gentzen's cut-elimination theorem, the equivalence of **G1** plus the cut-rule with Hilbert-style axiomatizations, and the completeness theorem for Hilbert-style axiomatizations. The direct proof is easy and does not involve the machinery of the cut-elimination theorem, which is really a separate matter.

where by hypothesis $C\mu\theta = B\mu\theta$, we can transform the derivations of the hypotheses using the induction hypotheses, to obtain a derivation terminating as follows:

$$\begin{array}{rl} A \to B, \Gamma \Rightarrow C & : \mu\theta \qquad (\to\Rightarrow) \\ A \to B, \Gamma \Rightarrow A & : \mu\theta \\ B, A \to B, \Gamma \Rightarrow C & : \mu\theta \end{array}$$

This still meets the condition required to be a Prolog derivation, namely $C\mu\theta = B\mu\theta$. That completes the proof.

5 The Program GENTZEN

In this section we exhibit a simple version of a Prolog program we call GENTZEN for a theorem-prover for the full intuitionistic predicate calculus. Like any Prolog program, GENTZEN determines a non-deterministic algorithm, as well as a deterministic algorithm that will be executed when the program is run. The emphasis in this paper is on non-deterministic GENTZEN. Questions of efficiency and completeness of non-deterministic GENTZEN will be taken up briefly in Section 9 and continued in another paper.

This program has the following properties:

• Restricted to Prolog sequents, GENTZEN's computations coincide with Prolog's deduction algorithm.

• GENTZEN can be applied to any sequent.

• It is sound: only intuitionistically valid formulae can be derived by GENTZEN, even non-deterministically.

• It is (non-deterministically) complete for intuitionistic predicate calculus: if a sequent has a **G** proof, the (nondeterministic version of) the algorithm will find it eventually. In other words, if GENTZEN terminates with failure, the sequent has no **G** proof.

• On input $\Gamma \Rightarrow \exists x A$, if there is a term t such that $\Gamma \Rightarrow A[t/x]$ is derivable, non-deterministic GENTZEN can find one. (It can also derive such formulae in case no such term t exists.)

Deterministic GENTZEN, like Prolog, is not complete. GENTZEN is really a family of programs, some of which include more "redundancy checks" than others. The role of redundancy checks is to prevent infinite loops in the deterministic execution of GENTZEN; they are irrelevant to non-deterministic GENTZEN. In general, the more redundancy checks, the slower GENTZEN runs, though of course there are efficient and inefficient ways of implementing a given redundancy check. These issues are discussed in Section 9. The program listed here contains no redundancy checks at all (except the obvious checks for cognate sequents one immediately above the other).

GENTZEN generalizes the Prolog program given for Horn clause logic in Section 3, so the reader is invited to study that much shorter program first. Before listing the program, we provide some information about its design.

The main predicate of the program is `derive((`$\Gamma \Rightarrow A$`))`, which constructs a derivation of the sequent $\Gamma \Rightarrow A$. If A contains Prolog variables X, these will be instantiated to object terms t and a **G** derivation of $\Gamma \Rightarrow A[t/X]$ will be constructed. The derivation trees themselves are not visible to the user; they are constructed only internally by Prolog. [16]

The predicate `prove(`A`)` means to derive the sequent $\Gamma \Rightarrow A$, where the axioms Γ are specified in an input file that has been consulted (see below); or if there is no such file, then it means to derive the sequent $\Rightarrow A$ with empty antecedent.

It is now time to raise the question: in which sequent calculus does GENTZEN construct a derivation? In fact there is a version of GENTZEN based on **G3**, a version based on **G**, and a version based on **G4**. The non-deterministic versions of these all turn out to be complete. The deterministic versions all solve all the examples in this paper. The **G3** version has the simplest program (because you don't have to worrry about when to delete hypotheses); but it is the **G4** version (or the **G** version) which imitate successful Prolog computations exactly. The labels `drop` in the program below are used to implement the omitting of formulae in the premise of rule $\rightarrow\Rightarrow$. You should ignore them at first reading.

Logical Notation in Prolog

GENTZEN uses the Prolog notation `(A,B)` for $A \wedge B$, and `(A;B)` for $A \vee B$. [17] Implication is written `A -> B`. [18] Negation is written `neg(A)` to distinguish it from Prolog's negation by failure, `not(A)`. Object variables (which can be quantified) are represented by Prolog atoms (except `true` and `false`). $\forall x A$ is written `all(x,A)` and $\exists x A$ is written `exists(x,A)`. As mentioned previously, we allow a more general syntax in which one may quantify over a list of variables all at once. The notation for this is, for example, `all([x,y,z],(a(x,y) -> b(z)))`. The list of object variables is required to be in lexicographic order, as defined by the Prolog system's primitive ordering of terms.

For reasons to be explained in the section on proofs by induction, we also allow λ-abstraction on formulas; the syntax is explained in that section, and the clauses of the program mentioning `lambda` and `ap` should be ignored for now.

Treatment of Variables

We distinguish *Prolog variables* (written as identifiers beginning with an uppercase letter) from *object variables* (treated as Prolog atoms, i.e. beginning

[16] Of course, if you want to see them explicitly, it is easy to add another argument to `derive`; in fact our first program did so. But you gain efficiency by leaving the proof-trees internal.

[17] We could have used `A  B` and `A / B` or even `A & B`; and we could have made the prover accept all these notations, but at a price in efficiency.

[18] Unfortunately the operator declarations are such that extra parentheses are required when formulae formed with binary operators are used as arguments: we need `prove((A -> B))`, not just `prove(A->B)`.

with a lower-case letter). Prolog variables should be thought of as ranging over terms of the object language (or sometimes, over other syntactic categories such as variables or even formulae). That is, they serve as what logicians call "meta-variables".

If the formula A contains any Prolog variables in the syntactic position of a term, the program attempts to instantiate them by terms. For example, the query:

```
prove(( a(0) -> a(X) ))
```

will succeed and return `X=0`. The query

```
prove(( a(0) -> exists(x,a(x)) ))
```

will succeed, but will return only `yes`. The query

```
prove(( exists(x,a(x)) -> exists(x,a(x)) ))
```

will succeed, returning `yes`, but the query

```
prove(( exists(x,a(x)) -> a(X) ))
```

will fail, because there is no term t such that $\exists x \mathtt{a}(x) \Rightarrow \mathtt{a}(t)$ is provable. Thus in practice, if you want an "instantiating term" t for a sequent $\Gamma \Rightarrow \exists x A$, you must first ask with a Prolog variable for x. If there is an instantiating term t, you will get it. If you then want to know whether the sequent is derivable (without an instantiating term), you ask again, this time with an explicit existential quantifier and an object variable. (Of course this double query method could be automated.)

Treatment of Axioms

Most problems in automated deduction involve a given set of axioms Γ, and a desired conclusion A. We then desire a derivation of the sequent $\Gamma \Rightarrow A$. As a matter of convenience, we want to enter the axioms Γ in a text file, and type only the desired conclusion A directly to GENTZEN. This is done by means of the predicate `axiom` of two arguments. The first argument is an (optional) name of the axiom. The second argument is the axiom itself.

Axioms should be *closed* formulae, i.e. should have no free variables, either object variables or Prolog variables. If you put a Prolog variable X in an axiom $B(X)$, and ask for a proof of $B(X) \Rightarrow A$, you are in effect asking if there is a term t such that $B[t/x] \Rightarrow A$ is provable. This may or may not be the case, but it is a different question from asking if $\forall x B(x) \Rightarrow A$ is provable.

The key to Prolog's efficiency, generalized

Note particularly the crux of the algorithm in the treatment of the rules $\Rightarrow \exists$ and $\forall \Rightarrow$, where Prolog variables are introduced for the instantiating term. When the computation reaches the `axiom` rule, unification will instantiate these variables, producing the desired terms efficiently. Note also the treatment of rule $\forall \Rightarrow$, in which we check whether the right branch is an axiom, and if it

is not, we select another rule to apply, rather than go 'up' the right branch further. Only if all else fails do we go up the right branch. This corresponds to Prolog's selection of a clause whose head unifies with the goal, and causes the program to exactly imitate Prolog's computation on Horn clause input. In the case of more general input, it often causes the quick and efficient selection of the correct axiom to use at the next step. As much as unification, this restriction on the search for a proof-tree is the key to Prolog's efficiency:

Don't look up the right branch of $\rightarrow\Rightarrow$ *unless nothing else works.*

Avoiding infinite regress

Kleene [1952], pp. 482, uses Gentzen's system **G3** to give a decision method for intuitionistic propositional calculus. The point of using **G3** (**G** would also do) instead of the system **G1** is that as we work 'upwards' in constructing a proof tree, no new copies of formulas already present are introduced by 'thinning' as in **G1**. This lets us bound the size of the (possible) proof tree. However, even in **G** it is possible that "loops" can occur in this process. For example, consider the sequent $a \rightarrow b, b \rightarrow a, a \Rightarrow a$. Prolog will loop on this example: to avoid the loop, you must write the input in a different order, for example $a, a \rightarrow b, b \rightarrow a \Rightarrow a$. GENTZEN will not loop on this particular example, as it checks for axioms before using $\rightarrow\Rightarrow$, but even this example illustrates the fact that infinite regress is possible in the process of constructing proof trees in **G** .

If one wants only 'parlor algorithms', it is easy to avoid loops by stopping whenever a 'redundancy' is created, i.e. a formula is generated that already occurs 'below' (on the same branch). However, it is difficult to check for redundancies efficiently. In the case of Prolog, it would change a linear algorithm to $O(n \log n)$ at best. So Prolog omits "loop checks"; and following this lead, we present here a version of GENTZEN with no redundancy checks. This leaves it up to the user of GENTZEN (like the user of Prolog) to avoid infinite regress by choosing a suitable ordering of the axioms before presenting them to GENTZEN.

See Section 9 for further discussion.

Can I run GENTZEN *on my machine?* The program runs in Arity Prolog, version 5.0x, on the IBM AT. It is written in standard Clocksin and Mellish Prolog, except for the operators `==` and `\==`. The operator `X == Y` tests whether `X` and `Y` are identical terms, even if they contain free variables, without instantiating any of the free variables in those terms. The operator `X \==Y` is its negation. So the program should run in any Prolog supporting these operators. Perhaps one should also worry about the relative precedences of the logical operators, which may vary between implementations of Prolog for all the author knows.

We present below a version of GENTZEN based on the system **G3** instead of on **G**. For simplicity, we present the version with absolutely no redundancy checks, and no attempt at controlling the search for a derivation except the one needed to explain Prolog. This simple version, however, is included in entirety.

If you are a Prolog programmer, you can supply the utilities described in Section 3, and then run the program whose main part is listed below. You can verify for yourself that it solves the examples given in later sections. Just consult your axiom file and give the query `prove(`*Goal*`)`. Anyone seriously interested in running GENTZEN should request source code for this and more complicated (but more efficient) versions of GENTZEN from the author, as the version presented here is really just a toy (albeit a working toy) whose purpose is to demonstrate the principles on which GENTZEN is based.

We list here only the main predicates `prove` and `derive`. A short description of some of the utility predicates used has been given already in Section 3. The predicate `pieces` is declaratively identical to `append`, but is used to break a given list into pieces (in different ways on backtracking) rather than to append two given lists.

GENTZEN *Partial Program Listing*

```
% direct implementation of Gentzen sequents
% without explicit proof trees */
% No redundancy checks at all, for speed and simplicity
% No control of search beyond the
% (right-branch of -> => an axiom) principle
% Author:  M. Beeson
% last edited 6.27.88
% original date of this version 5.18.88
% similar program with explicit proof-terms and equality, 9.30.87
 :- op(1160, xfx, =>).
%infix, non-associative, binds looser than ',' and ';'.
 prove(A):-
   findall(B, axiom(Name,B) ,Gamma),
   derive(( Gamma => A)).
 axiom((Gamma => A)):-            % A = false is legal
   member(X,Gamma), unify(X,A).
 % for finitely axiomatized theories, "member(A,Gamma)" would suffice;
% but this allows unification to select an instance of an axiom
schema
 derive(( Gamma => A )):-  axiom((Gamma => A)).
 derive(( Gamma => T=T )).
       %this accounts for Prolog's treatment of equality
 derive(( Gamma =>  A->B )):-       %rule => ->
    derive(( [A|Gamma] => B )).
derive(( Gamma => (A,B) )):-       %rule  => &
    derive(( Gamma => A)),
    derive(( Gamma => B)).
 derive(( Gamma => (A;B) )):-       %rule => ';'
   derive(( Gamma => A )).
 derive(( Gamma => (A;B) )):-       %rule => ';'
   derive(( Gamma => B )).
```

```
derive(( Gamma => neg(A))):-       %rule => neg
  derive(( [A|Gamma] => false )).
derive(( Gamma => C )):-           %rule & =>
   pieces(First,[(A,B) | Rest],Gamma),
   append(First,[A,B|Gamma],Delta),
   derive(( Delta => C )).
derive(( Gamma => C )):-           %rule ';'=>
  pieces(First,[(A;B)|Rest], Gamma ),
  not(fmember(A,Gamma)),          %prevent cognates on left branch
  not(fmember(B,Gamma)),          %prevent cognates on right branch
  append( First, [A|Rest], LeftHyp),
  derive(( [A|Gamma] => C )),
  append( First, [B|Rest], RightHyp),
  derive(( [B|Gamma] => C )).
```

Taking the right branch first as the following clause specifies is important when using this clause to instantiate a schema as in proofs by induction. On the other hand, one of the keys to the efficiency of GENTZEN is the use of `axiom` in the next clause instead of `derive`.

```
derive(( Gamma => C )):-          %rule -> =>
    memberandrest( drop(A->B), Gamma,Delta),
              %drop A->B, it came from (all =>)
    A \== C,             %prevent an obviously redundant proof
    axiom(( [B] => C )),         %[B] instead of [B|Gamma] is ok
    not(fmember(B,Gamma)),
              %prevent redundant proofs (cognates on right branch)
    derive((Delta => A).
derive(( Gamma => C )):-          %rule -> =>
    member( (A->B), Gamma),      %don't drop this occurrence of
A->B
    A \== C,             %prevent an obviously redundant proof
    axiom(( [B] => C )),         %[B] instead of [B|Gamma] is ok
    not(fmember(B,Gamma)),
              %prevent redundant proofs (cognates on right branch)
    derive((Gamma => A).
derive(( Gamma => C )):-              %rule  neg=>
  member(neg(A),Gamma),
  A \== C,                            %prevent immediate redundancy
  derive((  Gamma => A )).
derive(( Gamma => all([X|Rest],A) )):- %rule => all (list quantifier)
  gensymlist([X|Rest],x,Varlist),
          %make a list of new object variables
  list_fsubst(Varlist,X,A,AofVar),
  derive(( Gamma => AofVar )),
  not (member(Var,Varlist),fcontains([all([X|Rest],A)|Gamma],Var)).
       % restriction on variables
```

```
 derive(( Gamma => all(X,A) )):-  %rule => all (ordinary quantifier)
   gensym(x,Var),
   fsubst(Var,X,A,AofVar),
   derive((  Gamma => AofVar  )),
   not fcontains([all(X,A)|Gamma],Var).  % restriction on variables
 derive((  Gamma => C  )):-         %rule  exists => (list quantifier)
   member( exists([X|Rest],A), Gamma),
   gensymlist([X|Rest],x,Varlist),
         %make a list of new object variables
   list_fsubst(Varlist,X,A,AofVar),
   derive(( [AofVar|Gamma] => C )),
   not (member(Var,Varlist), fcontains([C|Gamma],Var)).
        % restriction on variables
 derive((  Gamma => C )):-     %rule  exists => (ordinary quantifier)
   member( exists(X,A), Gamma),
   gensym(x,Var),
   fsubst(Var,X,A,AofVar),
   derive(( [AofVar|Gamma] => C )),
   not fcontains([C|Gamma],Var).  % restriction on variables
                %rule => exist (list quantifier)
derive((Gamma => exists([X|Rest],A))):-
   findall(_,member(_,[X|Rest]),NewVarList),  %generate new variables
   list_fsubst(NewVarList,[X|Rest],A,NewA),
   derive(( Gamma => NewA )).
                %rule => exists (ordinary quantifier)
derive(( Gamma => exists(X,A) )):-
   fsubst(_,X,A,NewA),
   derive((Gamma => NewA )).
 derive((Gamma => B)) :-              %rule all =>  (list quantifier)
   member(all([X|Rest],A),Gamma),
   findall(_,member(_,[X|Rest]),NewVarList),  %generate new variables
   list_fsubst(NewVarList,[X|Rest],A,NewA),
   not fmember2(NewA,Gamma),
   ((functor(NewA,'->',_),
         derive(( [drop(NewA)|Gamma] => B ))
      );              %label implications with ‘drop’
    derive(([NewA|Gamma] => B))
   ).
 derive((Gamma => B)) :-     %rule all =>  (ordinary quantifier)
   member(all(X,A),Gamma),
   fsubst(_,X,A,NewA),
   not fmember2(NewA,Gamma),
   ((functor(NewA,'->',_),
         derive(( [drop(NewA)|Gamma] => B ))
      );           %label implications with ‘drop’
    derive(([NewA|Gamma] => B))
```

```
      ).
    derive( (Gamma => ap(lambda(X,A),T) )):-
      fsubst(T,X,A,AofT),
      derive((Gamma => AofT)).
    derive((Gamma => A)):-
      pieces(Firstpart,[ap(lambda(X,A),T)|Rest], Gamma),
      fsubst(T,X,A,AofT),
      append(Firstpart,[AofT|Rest],Delta),
              %replace ap(lambda(X,A),T) by AofT
      derive(( Delta => A )).
    derive(( Gamma => C )):-   %rule -> =>, going up right branch further
       memberandrest( drop(A->B), Gamma,Delta),
       A \== C,                          %prevent immediate redundancy
       not(fmember(B,Gamma)),
          %prevent immediate redundancy on right branch
       derive(( [B|Delta] => C )),  %Not just "axiom" but "derive"
now
       derive((Delta => A)).
    derive(( Gamma => C )):-  %rule -> =>, going up right branch further
       member( (A->B), Gamma),
       A \== C,                          %prevent immediate redundancy
       not(fmember(B,Gamma)),
          %prevent immediate redundancy on right branch
       derive(( [B|Gamma] => C )),  %Not just "axiom" but "derive"
now
       derive((Gamma => A)).
```

Note: In the above program, `gensym` is supposed to generate new variables. If your axioms contain variables such as `x17`, you should run `gensym` enough times first to be sure that the variables it generates will indeed be new. In practice it's simpler to avoid using such variables in the axioms.

6 McCarthy's sterilization example

John McCarthy has given the following example to illustrate the shortcomings of Prolog.[19] We will show that GENTZEN works the problem nicely. In order to help the reader understand how GENTZEN works, we will trace the execution of GENTZEN on this example.

```
∀ container( ∀ bug (in(bug,container) → dead(bug))      (sterile1)
                                        → sterile(container)
```

[19] See McCarthy [1987] where the problem is informally stated and the unsuccessful attempt to formalize it in Prolog is discussed. The axioms given here were written down by L. T. McCarty in a talk. Similar axioms (omitting `sterile2` and coalescing `heat1` and `heat2`) are in Miller [1988], p. 65; McCarty's work and Miller's work are discussed in Section 11.

$$\forall\ \texttt{bug(}\ \exists\ \texttt{container (heated(container)}\ \wedge\ \texttt{in(bug,container))} \rightarrow\ \texttt{heated(bug))} \quad \text{(heat1)}$$

$$\forall\ \texttt{bug(heated(bug)}\ \rightarrow\ \texttt{dead(bug))} \quad \text{(heat2)}$$

$$\forall\ \texttt{container (sterile(container)} \quad \text{(sterile2)}$$
$$\rightarrow\ \forall\ \texttt{bug (in(bug,container)}\rightarrow\ \texttt{dead(bug)))}$$

$$\texttt{heated(dish1)} \quad \text{(special)}$$

McCarthy points out that Prolog cannot represent this simple logic problem, let alone solve it, as it isn't formulated in Horn clause logic.

We shall show how GENTZEN solves this problem. The initial goal given to the program is `prove(sterile(dish1))`. First, the rule $\forall \Rightarrow$ is used to open up `sterile1`, replacing the bound variable `container` with a Prolog variable. Then rule $\rightarrow\Rightarrow$ applies, since the goal matches the "head" of `sterile1`. This generates the new goal $\forall$ `bug (in(bug,dish1)` $\rightarrow$ `dead(bug)`. Next GENTZEN uses $\forall \Rightarrow$ to open up the other universal axioms (and also the one which has already been used once), replacing the bound variables by Prolog variables. But none of the implications thus created can be used yet, so we reach the clause for $\Rightarrow \forall$. A new variable name `x1` is generated and we get the goal to prove `in(x1,dish1)` $\rightarrow$ `dead(x1)`. Then rule $\Rightarrow\rightarrow$ moves `in(x1,dish1)` to the antecedent, and we have to prove `dead(x1)`. Finally the goal matches the head of an implication in the antecedent, namely the one obtained by opening up `heat2`. Rule $\rightarrow\Rightarrow$ generates the new goal of proving `heated(x1)`. This matches the head of the implication obtained from `heat1`, and generates the new goal of showing that `x1` is in some heated container. Now rule $\Rightarrow \exists$ comes into play, replacing `container` with a Prolog variable and setting up the goal `heated(Container), in(x1,Container)`. Then $\vee \Rightarrow$ calls for proving `heated(Container)`. That goal, however, unifies with the axiom `heated(dish1)`, so `Container` is instantiated to `dish1`, leaving the goal `in(x1,dish1)` to work on. That formula, however, is in the antecedent, so the clause for `axiom` applies. GENTZEN now exits from all these recursive calls, back to the rule $\Rightarrow \forall$, where it still has to check the restriction on variables: indeed, $x1$ is not free in the antecedent at that point. That completes the proof.

It is interesting to note that the order of the axioms is important: if we put the axiom `heated(dish1)` last, then an infinite regress results. The system tries to prove `dish1` is heated by finding a container x such that `dish1` is in x and x is heated. This it will try to do by trying to find a container y such that x is in y and y is heated, and so on. This phenomenon is familiar from Prolog. In this case, it reflects a failure of the axioms to express all the information of

the informal problem: we should have used a typed system with types `bug` and `container`, or at least unary predicates *bug* and *container*, and made sure that `in`(x, y) holds only when x is a bug and y is a container. Similarly, the axioms above permit the deduction of `dead(dish1)`, which is surely not intuitive! The role of types will be discussed more in another section.

In the version of GENTZEN listed in Section 5, with no redundancy checks, it is important that the axiom "sterile2" come *after* "heat2". Otherwise GENTZEN tries to prove `dish1` is sterile by showing all bugs in it are dead, and then tries to prove that by showing it is sterile, entering a loop. One intermediate version of GENTZEN has redundancy checks only on rule $\forall \Rightarrow$. This version doesn't care where you put "sterile2", but it still must have "special" last. Its speed is intermediate between the version with no redundancy checks and full redundancy check.

7 Completeness of Non-deterministic GENTZEN

The program listing of GENTZEN, like any Prolog program, describes a non-deterministic algorithm, as well as a deterministic algorithm determined by the actual execution under Prolog. Just as all known metatheorems about Prolog concern the non-deterministic version, so our completeness theorem concerns non-deterministic GENTZEN. As in the case of Prolog, in the case of the actual running prover, it matters in what order you state your antecedent formulae. [20]

Note: In the theorems of this section, "GENTZEN" refers to the non-deterministic algorithm defined either by the listing of `derive` in Section 5, or by the similar but simpler programs based on **G3** or even **G1**, or by any more elaborate version of the program which improves on that listing's deterministic efficiency while retaining non-deterministic equivalence.[21] The general plan in developing GENTZEN as a practical theorem prover is to make modifications for efficiency which still preserve non-deterministic equivalence to the prototype `derive` studied here.

Theorem 4 (Soundness and Completeness of non-deterministic GENTZEN) *Let $\Gamma \Rightarrow A$ be a closed, intuitionistically valid sequent. Then* GENTZEN *answers the query* `prove`$(\Gamma \Rightarrow A)$ *by constructing a* **G** *derivation of the sequent.*

If the sequent is allowed to contain free variables $x = x_1, \ldots, x_n$, and if $X = X_1, \ldots, X_n$ are corresponding Prolog variables, then GENTZEN *answers*

[20] The classic example is the logic program for `member`, which has two clauses:

```
member(A,[A|X]).

member(A,[B|X]):- member(A,X).
```

Putting these clauses in the other order produces infinite regress in the computations generated by the query `member(a,X)`. This is equally true whether the clauses are given to Prolog or to GENTZEN as listed in this paper.

[21] Two versions of `derive` are non-deterministically equivalent, if they determine the same derivable formulae when the Prolog clauses are interpreted as the clauses of an inductive definition of "derivable".

the query `prove`$(\Gamma[X/x] \Rightarrow A[X/x])$ *by constructing terms* t *and a derivation of* $\Gamma[t/x] \Rightarrow A[t/x]$*, if it is possible to find such terms.*

Any answer to either of the above queries (for a sequent not containing equality) implies the construction of appropriate derivations, so GENTZEN *is sound.*

Corollary 1 (Classical Completeness) *Let* $\Gamma \Rightarrow A$ *be a closed, classically valid sequent, in which every atomic formula is doubly negated and in which* $\vee$ *and* $\exists$ *have been replaced by the classical equivalents not involving these connectives. Then* GENTZEN *constructs a* **G3** *derivation of* $\Gamma \Rightarrow A$*.*

Proof of Corollary. This follows from the theorem by Gödel's double-negation interpretation (Kleene [1952], p. 495, Theorem 60(d)).That completes the proof.

Proof of Theorem. The soundness of GENTZEN is proved by a straightforward induction on the length of (non-deterministic) Prolog computations of `derive`(Γ,A). There is one induction step corresponding to each clause in the program for `derive`.

Turning to the completeness, let $\Gamma \Rightarrow A$ be an intuitionistically valid sequent. By the completeness of Gentzen's cut-free rules (which can be proved either directly or using the cut-elimination theorem), there is an irredundant derivation of $\Gamma \Rightarrow A$. [22] It thus suffices to proceed by induction on the length of irredundant derivations, showing that every sequent with an irredundant derivation is proved by (non-deterministic) GENTZEN. We shall carry out the proof for the version of GENTZEN based on **G3**; the proof for the version based on **G** is only slightly more complicated, but we have given the program listing only for the version based on **G3**. Using the notion of *extended derivation* introduced in Section 3, we prove more: if there is an irredundant extended derivation of $\Gamma \Rightarrow A \quad : \theta$, where the variables free in the extended sequent are Prolog variables, and θ is the identity on variables not contained in $\Gamma \Rightarrow A$, then there is a Prolog computation of the query `derive(`$\Gamma \Rightarrow A$`)` such that the substitution produced by Prolog is more general than θ. This statement implies both parts of the completeness theorem.

If you seriously intend to follow the proof, you should read it with a copy of the program for GENTZEN at hand, as we will not repeat every clause when it is needed.

Basis case: $\Gamma \Rightarrow A \quad : \theta$ is an axiom in case there is a member B of Γ such that $A\theta = B\theta$. In this case the predicate `axiom`$((\ \Gamma \Rightarrow A\ ))$ will succeed, in view of its definition, with a unifying substitution more general than θ, since Prolog finds the most general unifier.

There is one induction step for each rule of **G3**.

[22] The notion of *redundant derivation* is defined in Kleene [1952], p. 482: it means that no two sequents on the same branch are *cognate*, i.e. have the same set of formulae in the antecedent and the same succudent. Evidently every derivation can be shortened to an irredundant derivation, since the rules of **G3** are construed to treat the antecedents as sets of formulae. An irredundant extended derivation is one whose associated derivation (obtained by applying the substitution at each line) is irredundant).

Rule $\Rightarrow \exists$: Suppose the last rule in the given **G3** derivation is rule $\Rightarrow \exists$. Then the last two lines of the derivation look like:

$$\begin{array}{rl} \Gamma \Rightarrow \exists x A & : \theta, x = X\theta \\ \Gamma \Rightarrow A[X/x] & : \theta \end{array}$$

where we may suppose x does not occur elsewhere in the sequent, and θ does not act on x. Then by induction hypothesis, the query `derive(`$\Gamma \Rightarrow$ `A[X/x])` succeeds with substitution more general than θ. Expressed more precisely, the right-hand side of the clause of GENTZEN for this rule succeeds. (Assuming, as we shall, that `fsubst` meets the specification given for it in the comments of the program.) Hence, the query `derive(`$\Gamma \Rightarrow$ `exists(x,A)` succeeds with substitution more general than θ.

All the rules involving list quantifiers are handled similarly to the cases of individual quantifiers, and will not be written out explicitly.

Rule $\forall \Rightarrow$: In this case we may assume the last two lines of the given derivation are of the form

$$\begin{array}{rl} \forall x A, \Delta \Rightarrow B & : \theta, x = X\theta \\ A[X/x], \Delta \Rightarrow B & : \theta \end{array}$$

where x doesn't occur elsewhere, and θ does not act on x.

The program works differently according to whether A is an implication or not; it labels implications with the prefix `drop`. It would be non-deterministically equivalent to the listing in Section 5 just to have two clauses, one applying to the case A is an implication, and the other the ordinary clause modelled on **G3**.[23] Since more clauses only make it easier to find proofs, we can for this proof just forget about the part of this clause involving `drop`. (After having checked its soundness.)

Note that since order is disregarded in the antecedent in the rules of **G**, the formula $\forall x A$ could actually be interspersed somewhere in Δ. The first line of the relevant clause of GENTZEN can choose (non-deterministically) this member of the antecedent (deterministically, it might choose a previous one!). The next line, involving `fsubst`, makes sure the variable is new, i.e. "x doesn't occur elsewhere". The next line rejects the derivation if the newly generated formula for the antecedent is already present; in that case the derivation would be redundant. Since we have assumed we have an irredundant derivation, this line will also succeed. By induction hypothesis, the last line succeeds with substitution more general than θ, completing this case. [24]

[23] Deterministically, that way of writing the program would give preference to cases of $\forall \Rightarrow$ in which the matrix is an implication, instead of just taking the leftmost formula $\forall x A$ in the antecedent.

[24] The line involving `fmember` prevents certain infinite regresses in the deterministic version.

Rule $\Rightarrow \forall$: In this case, we may assume the last two lines of the derivation are of the form

$$\begin{aligned} \Gamma \Rightarrow \forall x A \quad &: \theta \\ \Gamma \Rightarrow A \quad &: \theta \end{aligned}$$

where x does not occur free in Γ. Let x_{17} be the new variable produced by `gensym(x,Var)`. Renaming the free occurrences of the variable x in the derivation 'above' the root to be x_{17}, we obtain an irredundant derivation of $\Gamma \Rightarrow A[x_{17}/x] \quad : \theta$. Hence the call to `derive` in this clause of GENTZEN will succeed, by induction hypothesis. The last line of the clause is `not fcontains(`Γ`,`x_{17}`)`. Since Γ did not contain x free, it does not contain x_{17} at all. Hence this line succeeds, completing this case.

Rule $\exists \Rightarrow$: In this case, we may assume the last line of the given irredundant extended derivation is

$$\begin{aligned} \exists x A, \Delta \Rightarrow B \quad &: \theta \\ A, \Delta \Rightarrow B \quad &: \theta \end{aligned}$$

where x does not occur free in Δ or B, and (as in the case of rule $\forall \Rightarrow$) the formula $\exists x A$ may actually be interspersed in Δ. Non-deterministically, the first line of the relevant clause of GENTZEN can choose this member of the antecedent. Let x_{27} be the variable produced by the call to `gensym` in the second line of this clause; as above, rename the free occurrences of x above the root by x_{27}. Then x_{27} does not occur at all in Γ or B. By induction hypothesis, the recursive call to `derive` in this clause succeeds, with a substitution more general than θ. The last two lines will succeed since x_{27} doesn't occur in Γ or B, completing this case.

Rule $\rightarrow \Rightarrow$: The last lines of the given derivation are

$$\begin{aligned} A \rightarrow B, \Delta \Rightarrow C \quad &: \theta \\ (A \rightarrow B), \Delta \Rightarrow A \quad &: \theta \\ B, (A \rightarrow B), \Delta \Rightarrow C \quad &: \theta \end{aligned}$$

Of course the formula $A \rightarrow B$ in the antecedent might be interspersed in Δ; and as indicated by parentheses, it may even not occur in the premises. There are two clauses for this rule in the listing in Section 5, the first one checking for the case in which $A \rightarrow B$ has been labelled with `drop` when introduced by rule $\forall \Rightarrow$. Non-deterministically, we are free to ignore that clause; the second one suffices. We are assuming we have derivations of the premises; if the premises

actually omit the parenthesized $A \to B$, we can add it back to the antecedents of all formulae above this point, if necessary renaming some bound variables above this point to avoid violating the restrictions on variables. Hence we may assume that we have derivations of the premises with $A \to B$ *not* omitted; that is, that this inference follows the pattern of **G3**.

There are two cases, according as whether the 'right' (second) premise is an axiom or not. Let us first take the case in which it is, i.e. in which $C\theta = B\theta$, or possibly θ unifies C with some member of Δ. In the latter case, we would have found a successful computation of `derive` already under the clause calling `axiom` directly, so we may assume $C\theta = B\theta$. Now consider the computation by Prolog according to the first clause of GENTZEN for rule $\to\Rightarrow$. The first line (call to `member`) we may suppose has selected $A \to B$ from Γ, leaving Δ as the set of remaining formulae in the antecedent. If $A == C$, then we have a redundant[25] proof, since the left premise would be cognate to the root. Since by hypothesis the given derivation is irredundant, the second line `A \== C` succeeds. Since we have assumed $C\theta = B\theta$, the third line succeeds. Since the given proof is irredundant, the line `not fmember`(B,Γ) succeeds, for if B belongs to Γ then the right premise is cognate to the root. Finally, by induction hypothesis the last line, `derive`$((\Gamma \Rightarrow A\ ))$, succeeds with a substitution more general than θ, completing this case of this rule.

Now consider the second case of rule $\to\Rightarrow$, in which the right premise is not an axiom. This case corresponds to the second group of clauses of GENTZEN for this rule. (For efficiency's sake this clause is placed at the end of `derive`, but that is irrelevant to the present proof, which is about non-deterministic GENTZEN.) As above we may ignore the clause involving `drop` and assume that $A \to B$ is not omitted. The first line of this clause chooses a principal formula for the inference. The next two lines, which check for cognates on the left and right branch respectively, work as before. By induction hypothesis, the recursive call to `derive`$((\![B|\Gamma] \Rightarrow C\ ))$ succeeds, producing a Prolog computation of `derive(([B|Gamma] => C))` with a substitution θ_1 more general than θ. Using the definition of "more general", we can write $\theta = \theta_1\delta$ for some substitution δ. Applying the substitution θ_1 to the given derivation of $\Gamma \Rightarrow A \quad : \theta$, we obtain a **G3** derivation of $\Gamma\theta_1 \Rightarrow A \quad : \delta$. Applying the induction hypothesis to this derivation, we see that the Prolog computation of `derive`$((\Gamma\theta_1 \Rightarrow A\theta_1))$ will succeed with a substitution η more general than δ; thus $\delta = \eta\gamma$ for some substitution γ, so $\theta = \theta_1\delta = \theta_1\eta\gamma$. The substitution with which the computation of `derive`$((\Gamma \Rightarrow C))$ has progressed so far is $\theta_1\eta$, which is thus more general than θ. This completes the argument for this rule.

The cases corresponding to the other propositional rules are comparatively simple and are left to the reader to verify by inspection. That completes the proof.

[25] It might *become* redundant later, after a unification, even if this condition is not satisfied; but we have no way of checking for that now.

8 GENTZEN Extends Prolog

In this section, "GENTZEN" will be used generically to refer to versions either without redundancy checks, as in Section 5, or with redundancy checks. However, in this section we consider both deterministic and non-deterministic GENTZEN. When deterministic GENTZEN is considered, it is important that we consider a version somewhat more complicated than the listing exhibited in Section 5. That listing does not behave quite like Prolog, because it first instantiates a clause, and then retains the instantiated clause after it is used, which Prolog does not. To imitate Prolog, we must discard implications which result from instantiated clauses: we must actually omit the premises that **G** allows us to omit.

This version of GENTZEN is implemented by making the following changes to the listing in Section 5: (1) when rule $\forall \Rightarrow$ is used (in reverse) the new instance added to the antecedent is labelled with the functor `drop`. That is, the formula `drop(`A`)` is added to Γ rather than just A. (2) An extra clause for rules $\rightarrow\Rightarrow$ is added to the program, just before the clause in Section 5, which checks for formulae `drop`$(A \rightarrow B)$; it then takes such a formula for the principal formula of the inference, and "drops" the formula `drop`$(A \rightarrow B)$ from the antecedent, in accordance with the rules of **G**. (3) Similar clauses are added for rules $\wedge \Rightarrow$ and $\vee \Rightarrow$. The use of the functor `drop` saves repeated checking whether candidate principal formulae are substitution instances of other formulae in the antecedent. The resulting derivation will be in **G4** provided "droppable" formulae were not included among the axioms; if they were, however, some optional "drops" will be missed, and the derivation will be only in **G**.

We will prove that this version of GENTZEN imitates successful Prolog computations. This should be more or less obvious to the reader by now, although we give a proof below. The theorem, however, concerns only the *result* of the computation (that is, the answer substitution); while actually somewhat more is true: Not only does GENTZEN have the same *result* as Prolog, but the *computation process* is substantially identical. There is one difference, however: GENTZEN uses atomic clauses P first, rather than treating them as implications `true` $\rightarrow P$. If you want GENTZEN to *exactly* imitate Prolog, you have to write your atomic clauses as implications this way.[26] No doubt GENTZEN is slowed down slightly by the process of checking the program for atomic clauses instead of simply proceeding left-to-right.

Another point which may not be obvious is that the theorem only concerns *successful* Prolog computations. GENTZEN does not always imitate Prolog on *unsuccessful* computations: sometimes it terminates when Prolog goes into a loop. There are some invalid Prolog sequents on which Prolog does not terminate, even though they are purely propositional. For example, the sequent $a \rightarrow b, b \rightarrow a \Rightarrow a$. GENTZEN terminates and says `no` on such inputs. In this sense, GENTZEN *properly* extends Prolog: it improves on Prolog even in the propositional fragment. Similarly, there are *valid* Prolog sequents on which

[26] Prolog actually represents them as implications `true` $\rightarrow P$ in just this way in its internal database.

Prolog does not terminate (because the clauses are in the wrong order), such as $a \to b, b \to a, a \Rightarrow a$. GENTZEN terminates successfully on this example, so deterministic GENTZEN improves upon Prolog, even on the *valid* propositional fragment. See Section 9 for further discussion.

Theorem 5 (Extension of Prolog) *Non-deterministic* GENTZEN *extends non-deterministic Prolog. More precisely: Let the Prolog sequent* $\Gamma \Rightarrow A$ *correspond to the query* A *to the logic program* Γ*. Suppose Prolog answers* `yes` *with answer substition* θ *for the Prolog variables in* A *(if any). Then* GENTZEN *answers* `yes` *with the same answer substitution to the query* `derive(`$\Gamma \Rightarrow A$`)`*.*

Moreover, deterministic GENTZEN *(based on* **G4** *and running under Prolog) extends actual Prolog. That is, if* $\Gamma \Rightarrow A$ *is a Prolog sequent, and actual Prolog returns an answer substitution* θ *to the query* A *under the program* Γ*, then deterministic* GENTZEN *constructs an extended* **G4** *derivation of* $\Gamma \Rightarrow A \quad : \theta$*. This remains true for versions of* GENTZEN *including (various) redundancy checks.*

Proof. First we consider non-deterministic GENTZEN. We use the tool of extended **G** derivations introduced in the proof of completeness of Prolog.

Suppose the Prolog sequent $\Gamma \Rightarrow A$ corresponds to a successful query A to the (non-deterministic) logic program Γ. Note that Γ has no Prolog variables, though A may. Then (by Observation 1) there is an extended **G** derivation of $\Gamma \Rightarrow A \quad : \theta$, where θ is the answer substitution produced by Prolog. Removing any redundancies from this derivation, we may assume we have an irredundant extended **G** derivation of $\Gamma \Rightarrow A \quad : \theta$. By the completeness of GENTZEN, the query `derive(`$\Gamma \Rightarrow A$`)` succeeds, with the Prolog variables X of A instantiated by means of a substitution more δ more general than θ. By the soundness of GENTZEN, there is an extended **G** derivation of $\Gamma \Rightarrow A \quad : \delta$. Then by the completeness of Prolog, there is a Prolog computation answering the query A to program Γ by a substitution more η more general than δ, and hence more general than θ. But Prolog produces most general unifiers: hence $\eta = \theta$. Hence $\delta = \theta$ too. This completes the proof for non-deterministic GENTZEN.

Now consider the case of deterministic GENTZEN. To understand what has to be proved, consider what has been proved already in Section 4. There we considered a variant of GENTZEN obtained from the full **G4**-based GENTZEN by

(1) deleting from the program for `derive` all clauses except those corresponding to rules $\to\Rightarrow$, $\forall \Rightarrow$, and $\Rightarrow \exists$; and

(2) deleting the label `drop` from implications everywhere, i.e. in rules $\forall \Rightarrow$ where they are introduced and in rule $\to\Rightarrow$ where they are dropped.

(3) deleting the second group of clauses corresponding to rule $\to\Rightarrow$ (for going up the right branch), and retaining only the first clause of the two for going up the left branch; and

(4) in case of versions of GENTZEN with redundancy checks, deleting all redundancy checks.

We proved in Theorem 2 that this program corresponds to Prolog. Evidently in case the input is a Prolog sequent, only clauses corresponding to rules $\rightarrow\Rightarrow$, $\forall\Rightarrow$, and $\Rightarrow\exists$ can be used, so (1) is no problem.

(2) is no problem either: every implication has to have originated in a Prolog clause, and so got labelled with `drop` when it first appeared. Since every implication is labelled, we may as well drop the labels.

Since we only have to prove that GENTZEN succeeds when Prolog does, it does no harm whatever to add more clauses to `derive` at the end of the program; these can only result in GENTZEN succeeding when it otherwise would not. Hence (3) makes no difference, since the clause in question comes at the end of the program for `derive`.

Finally, consider (4). Adding a redundancy check can only improve GENTZEN's performance: no computations which succeeded without a redundancy check will fail when a redundancy check is included. Hence (4) makes no difference.

We have now proved that GENTZEN will succeed at least on those inputs for which the program in Section 4 succeeds; but by Theorem 2 this includes all cases in which Prolog succeeds. That completes the proof.

Remark: There are a number of more elaborate versions of GENTZEN not discussed in this paper; but they all work the same on the Prolog fragment, so the theorem applies to them all.

9. Deterministic GENTZEN *as a Theorem Prover*

Deterministic GENTZEN is incomplete. This section contains a preliminary discussion of the factors leading to this incompleteness, and of the tradeoff of incompleteness for efficiency. We describe an extension of GENTZEN that provides a decision method for intuitionistic propositional calculus.

We intend to show in the future by demonstration that control structures can be introduced into the framework provided by non-deterministic GENTZEN which produce a very efficient practical theorem-prover. A full discussion of these problems, let alone their solutions, is beyond the scope of this paper.

Using GENTZEN *for independence proofs*

GENTZEN attempts to generate a **G** derivation of the 'goal' sequent given it as input, by using the rules of **G** in reverse, proceeding 'upwards' from the goal, and relying on Prolog's unification to later determine the terms needed at rules $\Rightarrow\exists$ and $\forall\Rightarrow$. Non-deterministic GENTZEN can make "choices" about which rule to use next. Deterministic GENTZEN specifies the order of rules to be tried. However, Prolog's backtracking will eventually try all the rules, so long as GENTZEN does not go into infinite regress. More specifically, suppose we have completed a partial (extended) derivation up to a certain sequent $\Gamma\Rightarrow A$. We then choose a rule, generate its premises (possibly containing new Prolog variables), and try to complete the derivations of these premises. If we succeed, fine. If we fail, a new rule will be tried. The only danger to the completeness of deterministic GENTZEN is thus the possibility of infinite regress in the attempt

to complete the derivation of one of the premises. The following theorem makes this precise:

Theorem 6 *If deterministic* GENTZEN *terminates with answer* `no` *when asked to derive* $\Gamma \Rightarrow A$*, then* $\Gamma \Rightarrow A$ *is underivable in intuitionistic predicate calculus.*

Proof. Suppose GENTZEN answers `no` to the query `derive`($\Gamma \Rightarrow A$). There are no cuts or non-logical predicates in the clauses for `derive`. Hence, if GENTZEN terminates with failure, every branch of the search tree has been explored, so any solution that might be found by non-deterministic GENTZEN would have been found. Hence non-deterministic GENTZEN also fails on this query. By the completeness of non-deterministic GENTZEN, $\Gamma \Rightarrow A$ is underivable in **G** . That completes the proof.*Example.* GENTZEN returns `no` when asked to derive the sequent $\forall x(\ \mathtt{a} \vee \mathtt{b}(x)) \Rightarrow \mathtt{a} \vee \forall x \mathtt{b}(x)$. This sequent is therefore underivable. Compare Theorem 58(b), p. 487 of Kleene [1952]; GENTZEN proves this independence result and others like it automatically, by essentially the same proofs given in Kleene.

Causes of Infinite Regress

Remember that we are dealing with extended sequents, i.e. sequents that may contain Prolog variables and labeled with a substitution affecting those variables (and possibly some of the bound variables). Such a sequent $\Gamma \Rightarrow A \quad : \theta$ is called "valid" if there is a substitution δ refining θ (i.e. $\delta \geq \theta$) such that $\Gamma\delta \Rightarrow A\delta$ is valid.

There are two possible cases to distinguish: Case 1, the premise in question is not valid. Because there is no decision procedure for predicate calculus, infinite regress on at least some invalid sequents is absolutely inevitable. [27] Case 2, the premise in question is valid. Then we can distinguish three possible causes of infinite regress. First of all, a valid goal can generate an invalid subgoal, as in the case of a theorem $A \vee B$ where B is valid but A is invalid. If A causes infinite regress, then so will $A \vee$ `true`. Second, there can be infinite regress due to a loop: the same formula is generated again and again. This is a "redundancy" in Kleene's sense. We will discuss methods of eliminating this possibility below. The third cause is infinite regress without loops, as in the search for a proof of `member(a,X)` when the clauses for `member` are given in the "wrong" order.

Some Examples

We shall now give some examples to show that GENTZEN can go into loops and regresses in various ways, just like Prolog. Lest the reader form an ill opinion of GENTZEN at first acquaintance, we hasten to point out that this is not a bad thing: it is the price we pay for getting speedy performance most of the time, just as in Prolog. Just as Californians get used to earthquakes, Prolog programmers get used to occasional loops and regresses, and learn how to write

[27] In fact, it is possible to write a program which would take a theorem-proving program P as input, and produce as output a formula on which P would go into infinite regress.

programs that don't get out of hand. We want to use GENTZEN the same way.

Let us take Case 1 first. Is it possible to generate an invalid premise from a valid conclusion by the reverse application of one of the (extended) **G** rules? Yes, e.g. by rule $\Rightarrow \exists$ in case $\Gamma \Rightarrow \exists x A$ is provable but for all terms t, $\Gamma \Rightarrow A[t/x]$ is not. Also it is possible by a purely propositional rule: The valid sequent $\neg A \rightarrow \neg A$ can be derived from the invalid sequent $\neg A \rightarrow A$ by rule $\Rightarrow \neg$. Thus the possibility exists to improve GENTZEN by adding modules which check for invalidity, e.g. by semantic methods. [28]

Now consider Case 2. How can we save GENTZEN from infinite regress on valid formulae? First consider some examples where (deterministic) Prolog loops: $a \rightarrow b, b \rightarrow a, a \Rightarrow b$. In the attempt to prove b, the first clause $a \rightarrow b$ leads to an attempt to prove a. The third clause (which would settle the matter) is never reached, because the second clause is used first and leads to the new goal b, which is a previous goal and so leads to a loop. We call this example $loop(2)$. Similarly, one can construct $loop(n)$. For example, $loop(3)$ is $a \rightarrow b, b \rightarrow c, c \rightarrow a, a \Rightarrow b$. Note that GENTZEN will succeed on all these examples, because it tries the `axiom` rule before the rule $\rightarrow\Rightarrow$.

However, GENTZEN (as presented in Section 5) does not escape the Prolog phenomenon of loops. Something similar happens with the classical example of `member`, where if the definition of `member` is given with the two clauses in reverse order, the computation of `member(a,X)` will diverge, generating as first subgoal `X = [_|Y], member(a,Y)`. GENTZEN duplicates Prolog's behavior on this example. We recommend that the (serious) reader hand-simulate GENTZEN on this example, to see that cognate (extended) sequents do result one immediately above the other, but that they are not seen to be cognate when first generated, only after another unification takes place later 'up' the branch. Hence GENTZEN's checks for immediate redundancies do not find these cognates, permitting the computation to diverge like Prolog.

Loops can sometimes occur that involve quantifier inferences, although above we gave an examples of propositional looping and quantifier regress. For example, if the two clauses `sterile1` and `sterile2` are placed at the top in McCarthy's sterilization example (Section 6), then the attempt to prove `dish1` is sterile leads to an attempt to prove all bugs in it are dead, which leads to another attempt to prove it is sterile.

Moreover, such loops can occur of arbitrary length. For example, if we were to introduce another predicate `clean` and replace the axioms for `sterile` by three looping axioms saying that `sterile(x)` implies all bugs in `x` are dead, which in turn implies `clean(x)`, which in turn implies `sterile(x)`, McCarthy's example would fail. Similarly, we can hide loops of any given length inside universal quantifiers.

[28] In the dawn of automatic theorem-proving, Gelernter used such a method for geometry: Does this premise hold in the diagram? Since that time there have been other uses of semantic methods. The use of Gentzen sequents provides a clear opportunity for integrating semantic methods into an otherwise purely syntactic prover.

Redundancy Checks

It is easy to supply GENTZEN with a full redundancy check. All we have to do is add another parameter to `derive`, so that `derive(Avoidlist,Sequent)` constructs a **G** derivation of `Sequent` which does not contain any sequent in the input list of sequents `Avoidlist`. Then we define `prove(Sequent):-derive(Sequent,[])` (in the case of no axioms), and modify the clauses of `derive` so that the current goal is added to the front of `Avoidlist`. This algorithm has been implemented, and of course it eliminates the examples of incompleteness given under Case 2 above.

Such a scheme will be expensive, however: it will cost $O(d)$, where d is the depth of the proof tree so far constructed, at each step. This will turn a linear algorithm into a quadratic algorithm, in case the input is restricted to Prolog sequents.

By keeping the list of formulae on the current branch in a sorted array, so access time is independent of the length, we can reduce the cost to $O(\log d)$ at each step. That will give an algorithm with speed $O(d \log d)$ on Prolog sequent input. This algorithm cannot be implemented in Clocksin-and-Mellish Prolog, which does not support arrays or constant-access-time lists in any form. A similar scheme using B-trees could be implemented in Arity Prolog, but this has not yet been done. While doing this, we may as well keep a record of goals attempted (successfully or not) so as to prevent duplication of effort.

There may well be interesting programs intermediate between GENTZEN as listed and the version with full redundancy check. One such possibility is suggested by the "tortoise and hare" technique for loop-checking in Prolog. See the discussion of van Gelder's work in Section 11.

Propositional Decision Procedure

Theorem 7 *Deterministic* GENTZEN *with full redundancy check provides a decision procedure for intuitionistic propositional calculus.*

Remark: The theorem implies that we can decide if an arbitrary sequent can be proved by propositional axioms alone, even if the sequent contains function symbols and quantifiers.

Proof. As proved in Kleene [1952], p. 485, there is a bound on the depth of any branch of a partially constructed proof tree, since the sequents on the branch are all subformulae of the root, and a propositional formula has only finitely many subformulae. Hence infinite regress is impossible, and GENTZEN must eventually (by Prolog's backtracking) examine all possible ways of constructing an irredundant proof, finally either finding one or proving by its failure after a finite time that no irredundant proof exists. That completes the proof.

Remark: With the $O(n^2)$ implementation of full redundancy check, the sterilization example in the next section runs about six times slower than with no redundancy check. But: it runs no matter how you order the axioms.

Question: Implementing a full redundancy check makes the worst-case speed of the propositional decision procedure $O(\log n2^n)$, where n is the length of the input formula. One may think the exponential makes the log hardly worth worrying about; but all the same there is a theoretical question whether it can be eliminated. Does anyone know a decision procedure for propositional intuitionistic calculus which is faster than $O(\log n2^n)$? Is the validity problem for intuitionistic propositional calculus in Co-NP?

GENTZEN *as a Theorem Prover*

In addition to redundancy checks, we have made other improvements to GENTZEN, which for reasons of simplicity we have not included in the listing in Section 5. The general plan in improving GENTZEN's performance is to make restrictions on the application or order of application of the (reverse) rules, in such a way that the completeness of non-deterministic GENTZEN is not affected, but the performance of deterministic GENTZEN is improved. We have had considerable success in determining the proper control structures: for example, Schubert's Steamroller[29] can be solved by essentially the most efficient sequence of deductions, except for some duplicated derivations. (This takes 85 seconds in interpreted Prolog on an IBM AT; allowing a factor of 10 for compilation and a factor of three or four for the slow hardware, this is approximately state-of-the-art.) See Section 11 for a comparison with SATCHMO. The control structures involved will be sketched below.

The essence of the difficulties in using GENTZEN as it stands in section 5 can be seen by trying it on a problem like Schubert's Steamroller, in which there is a lot of branching in the proof construction process. Since GENTZEN proceeds depth-first, when confronted with several possible ways to proceed, it chooses the first and then fights to the death to prove the theorem that way, never stopping to consider that another choice might be much easier. Others have met the same fundamental problem, and tried to solve it by limiting the resources to be expended on a given branch, for example by "iterative bounded depth-first search", etc. Our idea is instead to limit the *means* which can be applied. We define `derive1` which (1) uses only the minimal-logic form of rule $\neg \Rightarrow$, (2) only the right-branch-an-axiom case of rule $\rightarrow\Rightarrow$, and (3) does not use rule $\vee \Rightarrow$. When faced with a choice, we use only `derive1` before going on. When `derive1` finally fails, we use another case of rule $\rightarrow\Rightarrow$: this time we require the *left* branch to be an axiom; and we also use $\vee \Rightarrow$. When these two rules can do no more, we start over with `derive1`, and so on. Only if all else fails do we use the full intuitionistic $\neg \Rightarrow$ or $\rightarrow\Rightarrow$. These rules almost always lead us down the garden path, except on examples concocted specifically to need them.

The Steamroller problem seems intuitively to be solved by a combination of backwards and forwards reasoning. It seems that the backwards reasoning corresponds to the right-branch-an-axiom case of $\rightarrow\Rightarrow$, and the forwards reasoning is the left-branch-an-axiom (or conjunction of axioms) case. GENTZEN (in this version) uses Prolog-style backwards reasoning till it does no more, then

[29] This is a standard "benchmark" problem for theorem provers. See Stickel [1986].

reasons forward to generate new facts that can be used for another round of backward reasoning, and so on.

Types in GENTZEN

We have also extended GENTZEN (in theory, but not yet in implementation) to various type theories. Analogues of the theorems proved here for first-order logic also can be proved in some of these situations; certainly they are without complication as long as the language does not include some form of λ-calculus, e.g. in a simple many-sorted predicate logic. We view the extensions to type theory as vital for a practical proof-checker or proof-finder, but a description of this work is beyond the scope of this paper. See Constable *et. al.* [1986], Feferman [1985], Huet [1987], and Martin-Löf [1984] for descriptions of the kinds of type theories to which GENTZEN can be extended, and Paulson [1986] for a type-theoretic theorem-prover based on natural deduction.

Equality in GENTZEN

As presented here, GENTZEN has no more ability to do equality reasoning than Prolog does. However, one can add clauses to `derive` with the head `derive((Gamma => X=Y))`, which embed in GENTZEN the equality reasoning mechanisms of your choice. That is, the framework of GENTZEN permits the natural integration of logical theorem-proving with, for example, rewrite rule techniques or even symbolic computation in the style of MACSYMA. These matters are also beyond the scope of this paper.

Using the Prolog Database to Store the Antecedent

One would like not to have to use an extra predicate `prove` to give GENTZEN a goal; one would like to type it directly to the Prolog prompt, and generally mix quantifiers and implications at will into Prolog programs. One needs the source code to Prolog to arrange that; but one may try to do it by using the Prolog database to store the antecedent formulae. Since this paper was first written, there have been several experimental versions of GENTZEN, testing different ways of storing the antecedent. In practical theorem-proving, the antecedent will be very long, including all the non-logical axioms and previously-proved theorems. The antecedent must, for efficiency, be stored in a fixed location rather than passed as a parameter to each call to the theorem-prover. Another aim of these experimental versions is to take as much advantage of the compiled inference supplied by the underlying Prolog as possible. A desideratum is that Prolog sequents ought to be derivable in GENTZEN as fast as they are when expressed directly in Prolog (at least, when GENTZEN is not using an occurs check). The program MATHPERT, designed for learning calculus, trigonometry, and algebra, incorporates one of these versions of GENTZEN to handle the logical aspects of calculus. For more information on the application of GENTZEN to calculus (but not on implementation details), see Beeson [1989].

9 Automating proofs by induction with GENTZEN

If Prolog variables are used in places where formulas belong while stating axioms, the program is not confused. In fact, it sometimes works beautifully: the unification step at rule `axiom` selects the appropriate instance of an axiom schema. Thus Prolog variables can be used not only to range over terms, but also to range over formulas. We have made use of this feature of GENTZEN to automate some proofs by induction in number theory.

The first problem is to state the axiom schemata of induction in Prolog using Prolog variables for formulae. Note that the usual method of stating the axiom of induction is not in this form, since it mentions substitution. It is for this reason that we have introduced λ-abstraction into the syntax of first-order logic (following Church and Aczel). [30] We introduce a primitive application operator **ap**, and allow $\mathbf{ap}(\lambda\mathbf{x}.\,\mathbf{A}, \mathbf{t})$ to be a formula whenever A is a formula and t is a term, and specify that to derive such a formula is the same as to derive $A[t/x]$, we obviously obtain a conservative extension of the usual formulation of first-order logic. This extension of notation is accepted by GENTZEN. We then can state the schema of mathematical induction as

$$\mathbf{ap}(\lambda\mathbf{x}.\,\mathbf{A}, \mathbf{0}) \wedge \forall\mathbf{x}(\mathbf{A} \to \mathbf{ap}(\lambda\mathbf{x}.\,\mathbf{A}, \mathbf{s}(\mathbf{x})) \to \forall\mathbf{x}\mathbf{A}$$

in which we can understand A as a Prolog variable. (In Prolog syntax, we write $\lambda x.\,A$ as `lambda`(x, A), and we write $\forall x A$ as `all`(x, A).)

We can now write down the axioms of Peano arithmetic (except the equality axioms) in a finite list, with one entry for the axiom of induction. Equality is a distraction to the point of this section, which can be made sufficiently well using only the single equality axiom $x = x$. This axiom is enough to illustrate proofs by induction. For example, in intuitionistic number theory the decidability of equality $x = y \vee \neg x = y$ has to be proved by a double induction, first proving $x = 0 \vee \neg x = 0$ by induction on x, then proceeding by induction on y. GENTZEN succeeds nicely in automatically generating the required instances of induction and finding this proof (which is rather tricky for advanced undergraduates) automatically.

To some extent, our success with induction is pure good fortune. Not every axiom schema can be treated in this way. For example of one that cannot, consider the axiom schema of the law of the excluded middle $A \vee \neg A$. You can ask GENTZEN to derive $\neg\neg a \to a$ from this axiom schema; but unfortunately it can't do it. The unification part of GENTZEN settles too soon on an instance of the schema, and never can find the right one.

[30] Church and Aczel were following Frege, who regarded a formula $A(x)$ as a propositional function. The quantifier $\forall$ then is a functional which applies to functions. What we usually write $\forall x A$ is really $\forall(\lambda x.\,A)$.

10 Relations to the Literature

We have certainly not worked in a vacuum; many of the ideas in this paper are "in the air". There are a number of connections of this with the work of others. As we found out only after writing GENTZEN, Felty and Shankar have written quite similar programs; the main ideas of using Gentzen sequents for a theorem prover and using Prolog's unification to find the terms needed, had already been anticipated by Bowen and found by Felty and Miller; the idea of using **G3** instead of **G1** was found more of less simultaneously by Shankar (but not by Felty and Miller); so perhaps the only really new ideas in this paper are (1) the explanation of Prolog in terms of Gentzen sequents, and (2) the rule *don't go up the right branch of a left implication unless all else fails*. The former suggested the latter, which seems to be the final key, after the other ideas mentioned, to an efficient extension of logic programming to all of first-order logic. In this section, we try to trace the connections with the work of others (given in alphabetical order). Apologies in advance to those whose work we may have omitted or mis-described.

Bledsoe

Bledsoe and his associates have implemented a theorem prover based on "natural deduction"; but more precisely, I believe it is actually based on a sequent calculus, using the cut rule only for the use of lemmas. They have added a number of other techniques to the prover, notably rewrite rules, variable shielding, and inequality chaining. I do not know how the search strategy used compares with that of GENTZEN.

Bowen

K. Bowen [1980] gives an explanation of Prolog in terms of Gentzen sequents. His work, like ours, uses unification to construct the terms needed by the rules $\forall \Rightarrow$ and $\Rightarrow \exists$. This is the main point of similarity.

However, his analysis differs from ours in several essential points. First of all, he does not analyze Prolog in terms of cut-free derivations, but in terms of derivations using only the cut-rule. He considers Gentzen derivations from "sequent axioms", and instead of considering the Prolog program as the antecedent of a sequent and the query as the succudent, he considers the query as a sequent and the program as a set of sequent axioms. Prolog's computation is then mimicked by the cut rule. This analysis does not permit a useful generalization to first-order logic.

Bowen does, however, give an algorithm for generating Gentzen proofs of an input sequent. The second main point of difference between Bowen's work and GENTZEN is that his algorithm (he calls it "reverse2") proceeds to generate *all* branches of a (prospective) proof tree, upwards from the goal sequent. The essential restriction that the exploration of the right premise of rule $\rightarrow\Rightarrow$ is not made in Bowen's work. His algorithm proceeds in breadth-first fashion, developing all branches simultaneously. This prevents the construction of later

("right-hand") branches from benefitting from unifications made deep in the left-hand branch. GENTZEN, by contrast, proceeds depth-first, like Prolog.

Moreover, Bowen's algorithm seems to depend on the idea stated in his Lemma (p. 9) that the rules of Gentzen's calculus are arbitrarily permutable. As we have seen in Section 2, this is not the case. Bowen's algorithm is thus in need of a more precise definition; as it now stands it will generate false "proofs" and fail to prove many valid sequents. In particular, this algorithm has surely never been implemented and tested.

A relatively minor point is that Bowen uses **G1** instead of **G3** or a hybrid system like **G** , and thus cannot obtain a decision procedure for propositional calculus.

Boyer and Moore Boyer and Moore's famous theorem-prover ([1979]) incorporates special heuristics for finding the right instance of mathematical induction needed to prove a given theorem. These techniques are more sophisticated than the unification-based choices made by GENTZEN. Like other "high-level" heuristics, these could be added to a GENTZEN-based theorem prover.

Feferman

Feferman [1975], [1979] (see also Beeson [1985]) has given theories which were originally presented as theories of operations and "classes", but which are in the present context more usefully thought of as type theories with variable types in which logic is not explicitly reduced to type theory. Feferman [1988] develops versions of the systems more explicitly suited to this viewpoint, and containing the polymorphic λ-calculus of Girard and Reynolds. GENTZEN can be extended to a theorem prover for this kind of system. There is one additional feature of the logic of these systems: they admit possibly undefined terms ("partial terms"). The necessary modifications to the logic are given in Beeson [1985], p. 97; it is not difficult to adapt GENTZEN to this logic.

Gabbay and Reyle In their papers [1984] and especially [1989], Gabbay and Reyle describe a theorem-prover for intuitionistic logic which is based on a sequent calculus.

van Gelder

van Gelder [1987] has given an interesting method of "loop detection" in Prolog, known as the "tortoise and hare" technique. In our context this appears as a method for detecting certain kinds of redundant proofs in linear time, instead of time $O(d \log d)$ or $O(d^2)$ as in Section 9. The method does not detect all redundancies, and since it was developed in connection with Prolog, it is concerned only with redundancies involving the $\rightarrow\Rightarrow$ rule. Perhaps it can be generalized to a larger fragment of logic. In any case, it would be of interest to add such techniques to GENTZEN, producing fast-running versions with at least some redundancy checks.

Girard

Girard's influential book [1989] contains the explanation of Prolog in terms of the cut-rule discussed under Bowen, above.

Hällnas and Schroeder-Heister

Their technical report [1987] contains a proof-theoretical explanation of Prolog similar to Bowen's, discussed above.

Hayashi

Hayashi [1987] has built a proof-checker PX for a version of Feferman's systems, writing in LISP. The logic of PX is based on natural deduction, rather than on Gentzen systems. PX can handle the logic of partial terms discussed above. A proof-finder such as GENTZEN can be used in a proof-checker, to increase the "step-size" of the proofs that can be checked: the lines of the proof to be checked are given as successive goals to the proof-finder. It would be interesting to see if the algorithm of GENTZEN could increase the power of a proof-checker like PX; but this will in practice require re-writing a system like PX based on Gentzen sequents instead of natural deduction, and in Prolog instead of LISP.

Lifschitz

Lifschitz [1986] has given a characterization of circumscription in terms of logic programming (or, as he would put it, a characterization of logic programming in terms of circumscription). His work applied to the case when the circumscribing formula can be expressed in Horn clause logic extended to "stratified" occurrences of negation. Perhaps there is an extension of his theorem to more general cases if negation is not treated as negation-by-failure; this is a topic for further research.

Manthey and Bry

Manthey and Bry [1988] describe a theorem prover called SATCHMO. It works on the "clausal fragment": in sequent calculus terms, it works on sequents with atomic succudent, and whose antecedent formulae are either atomic, or implications whose left side is a conjunction of atomic formulae and whose right side is a disjunction of atomic formulae. Consequently quantifier rules never come into play. As discussed in Section 9, on this fragment we can implement GENTZEN to use the Prolog database to store the antecedent formulae. It then turns out that GENTZEN, with the control structure sketched in Section 9, exactly reproduces the action of SATCHMO. This is somewhat remarkable, as the creators of SATCHMO thought of SATCHMO's computations as a model-construction process; but proof theorists have long known that the proof of the completeness theorem is a process which either constructs a model or a cut-free proof, so you can view it either way. The proof-theoretic framework of GENTZEN permits a unified understanding of the two processes in SATCHMO, whose interaction was previously a bit mysterious (at least to the author). In particular, GENTZEN with the control structure described in Section 9 proves the Steamroller very rapidly, and by essentially the same process as SATCHMO.

McCarty

L. McCarty [1988] has given a semantics of Prolog in terms of partial Kripke models, and an extension of Prolog to a fragment of intuitionistic predicate calculus he calls "clausal intuitionistic logic", which generalizes Prolog by permitting negation and universally quantified implications in the "body" of a (generalized) clause. McCarty [1988a] gives a tableau proof procedure for this fragment. The procedure has a similar flavor to GENTZEN, in that there are two kinds of variables (McCarty writes $?x$ for what correspond to our Prolog variables), and bindings are determined later and propagated down. There is surely a close connection between this tableaux method and GENTZEN, but it works only for the fragment mentioned.

L. McCarty (not yet published) uses intuitionistic negation in combination with negation by failure to define a form of "default reasoning": a "default rule" is one of the form $B \wedge \texttt{not}(\neg A) \rightarrow A$. In words: B normally implies A, i.e. it does unless we have evidence that A is false. That is, A is true by default if the condition B holds and at present we can't refute A. Since GENTZEN provides efficient means to handle intuitionistic logic (including negation), it would be interesting to develop a default reasoning system based on GENTZEN's inference mechanisms and McCarty's ideas on default reasoning.

Miller et. al.

Miller *et. al.* [1987] define the "hereditary Harrop formulae". This fragment is essentially defined so that when analyzed as Gentzen sequents, the reverse proof process never leads to any connective other than $\forall$ or $\rightarrow$ in the antecedent. (Thus other connectives are barred from the left part of implications also in the succudent.) The language λ-prolog is a logic programming language based on the hereditary Harrop fragment of (higher-order) logic. In this language, the quantifiers are represented as a composition of functionals $\forall$ and $\exists$ with λ-abstraction, as described in Section 10 of this paper. For this restricted fragment, the proof process described on p. 64 of Miller [1988] is essentially that of GENTZEN, since the formulae and the proof process are so restricted that the process amounts to using only those cases of rule $\rightarrow\Rightarrow$ in which the right branch is an axiom. Miller says that this process is non-deterministically complete for the hereditary Harrop fragment of intuitionistic logic. This result (at least for first-order hereditary Harrop formulae) is a special case of the completeness of non-deterministic GENTZEN.[31]

Felty and Miller [1988] report on the implementation of theorem provers (for all of logic, not just the hereditary Harrop fragment) in λ-Prolog[32]

[31] While neither is directly relevant to our work, we note that the fixed-point Kripke model semantics of Miller *et. al.* [1987] is extremely similar to McCarty [1988].

[32] They argue that ordinary Prolog is insufficient for writing theorem-provers, because of the difficulties of handling bound variables. They say that encoding formulas as first-order terms as GENTZEN does is "unnatural and spoils the elegance with logic programming offers". We disagree wholeheartedly; but we do agree that is (also) elegant to decompose quantification into a functional and λ-abstraction. Besides, from the point of view of efficiency, it is better

The implementation of Gentzen's calculus described in Felty and Miller [1988] is similar to the **G1** version of GENTZEN, but without the separation of the clauses for the rule $\rightarrow\Rightarrow$ into two cases, implementing the principle *don't go up the right branch of a left implication unless all else fails*. In fact, on p. 70, they admit "Another aspect of these theorem provers not yet considered is control." Then on p. 71, they mention the problem which led us to use **G3** or **G** instead of **G1**. Another minor difference is that they keep proof-terms explicit, rather than letting Prolog construct them only implicitly; this is a duplication (since Prolog constructs them internally anyway) and slows the prover down. (Of course, for proof *checking* you may need to keep the terms explicit.)

Nadathur

I have been told that Nadathur's Ph. D. thesis [1987] contains a proof of a theorem closely related to the "Main Lemma" of Section 4 of this paper. I have not yet seen the thesis.

NuPrl

The rules of NuPrl (Constable *et. al.* [1986]) are best understood as an extension of Gentzen's **G1** (with cut) to a typed formalism as discussed above. The cut rule is NuPrl's *SEQUENCE* rule. In spite of a superficial similarity to natural deduction, the function type rules clearly mimic Gentzen's rules. NuPrl is designed primarily as a proof-checker, not as a proof-finder. However, the use of "tactics" permits the user to write proof-finding algorithms in the programming language ML. One such tactic, `backchain_with` (pp. 202-203), is billed as being similar to Prolog. However, it "first breaks down the conclusion and then tries to back through each hypothesis in turn until it succeeds". This tactic also misses the point of stopping exploration of the right premise of rule $\rightarrow\Rightarrow$. Clearly, however, a tactic could be written for NuPrl which implements the algorithm of GENTZEN.

Paulson

Paulson [1986] describes a theorem-prover *Isabelle* which is based on natural deduction, as opposed to a sequent calculus such as GENTZEN uses. He has used *Isabelle* to implement Martin-Löf's constructive type theory.

Shankar

N. Shankar at Stanford has written a theorem-prover in LISP based on Gentzen's **G3**. This work is extremely closely related to ours. His prover differs from GENTZEN in only four ways:

- It does not incorporate the key provision, "don't go up the right branch of left implication unless all else fails", from which GENTZEN derives its ability to generalize Prolog.

to run a theorem prover written directly in Prolog than to run a theorem prover written in another language which itself is written in Prolog. Eliminate the middle-man. In my opinion, the main point of λ-prolog is the use of Huet's λ-calculus unification.

• It is written in LISP, not Prolog. This is inessential if one thinks of it as a theorem-prover, but it prevents adding one line (`derive(( Gamma => A )) :- call(A).`) to convert it into a generalized logic-programming language.

• It does, however, improve upon GENTZEN by implementing an algorithm for early detection of violation of the "restriction on variables" at inferences by rules $\Rightarrow \forall$ and $\exists \Rightarrow$.

Here is a more complete explanation of the last point: Let P denote a node in the proof where a "restriction on variables" has to be met. The variable to be introduced must not occur in the terms in the rest of the formula, but those terms are still under construction and will not be settled upon until unification takes place at the axioms. GENTZEN does not check the restriction on variables until it has already constructed a candidate proof 'above' P, which will then be tested for whether it meets the restriction on variables. Shankar's prover, by contrast, labels every formula with a list of variables which must be avoided. Hence if there are several branches above P in a derivation, his prover can detect after the first branch that the restriction on variables will be violated, while GENTZEN does not detect it until later, when all recursive calls to `derive` have been completed and execution returns to the point P. Here is a concrete example where this happens:

$$\forall x(\mathtt{a} \vee (\mathtt{b}(x) \wedge \mathtt{c}(x))) \Rightarrow \mathtt{a} \vee \forall x(\mathtt{b}(x) \wedge \mathtt{c}(x))$$

This is an unprovable sequent, but Shankar's prover discovers that fact sooner than GENTZEN, as follows. When we attempt to prove this sequent, it soon becomes apparent that you must 'open up' the left side first. Then you drop the $\forall x$ on the left and replace x with a Prolog variable X. Then you use $\vee \Rightarrow$ and soon come to the goal

$$(\mathtt{b}(X) \wedge \mathtt{c}(X))) \Rightarrow \forall x(\mathtt{b}(x) \wedge \mathtt{c}(x))$$

Then you use $\Rightarrow \forall$ to introduce a new variable x_1 subject to the restriction that x_1 is not free in X. Then you try to verify that from the given antecedent, both $\mathtt{b}(x_1)$ and $\mathtt{c}(x_1)$ are provable. Shankar's prover will realize after attempting the first of these two that it is impossible to meet the restriction on variables, but GENTZEN doesn't realize it until both attempts have resulted in candidate proofs.

It is not difficult to add Shankar's idea to GENTZEN.

Stickel

Stickel [1986a, 1988] describes his "Prolog Technology Theorem Prover" (PTTP) which is "an extension of Prolog that is complete for the full first-order predicate calculus". PTTP differs from GENTZEN in that (1) it is not designed for intuitionistic logic; (2) it is not intended to provide an *explanation* of Prolog; (3) it is not based on sequent calculus. It is similar to GENTZEN in that as a practical theorem prover, it implements a Prolog-like depth-first search strategy. It incorporates two additional features which can easily be added to

GENTZEN: iterative bounded depth-first search (to stop infinite regress), and unification with occurs check.

Thistlewaite, McRobbie, and Meyer

In their book [1988], they discuss the use of finite models to prune the search tree in propositional non-standard logics. The point is that if one had a fast way of detecting (some) non-theorems, one could save the effort of trying fruitlessly to construct that branch of a proof tree when a non-theorem is generated as a goal. The difficulty is that model-testing isn't fast; but these authors show that it is sometimes useful anyway.

References

Beeson, M. [1985], *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin/ Heidelberg/ New York (1985).

Beeson, M. [1989], Logic and computation in MATHPERT: An expert system for learning mathematics, in: Kaltofen and Watt (eds.), *Computers and Mathematics '89*, pp. 202-214, Springer-Verlag, New York/ Heidelberg/ Berlin (1989).

Bishop, E. [1967], *Foundations of Constructive Analysis*, McGraw-Hill, New York (1967).

Bledsoe, W. W. [1984], Some automatic proofs in analysis, in: *Automated Theorem Proving: After 25 years*, ed. by W. W. Bledsoe and D. W. Loveland, A.M.S. *Contemporary Mathematics* series, vol. **29**, Providence, R. I. (1984).

Bowen, K. [1980] Programming with Full First Order Logic (dissertation), School of Computer and Information Sciences, Syracuse University, November 1980.

Boyer, R., and Moore, J. [1979] *A Computational Logic*, Academic Press, New York (1979).

Clocksin and Mellish [ ], *Programming in Prolog*, Springer-Verlag, Berlin/ Heidelberg/ New York (1981).

Constable, R. L. [1986], *et. al.*, *Implementing Mathematics with the NuPrl Proof Development System*, Prentice-Hall, Englewood Cliffs, N. J. (1986).

Feferman, S. [1975], A language and axioms for explicit mathematics, in: *Algebra and Logic*, Lecture Notes in Mathematics **450** 87-139, Springer-Verlag, Berlin (1975).

Feferman, S. [1979], Constructive theories of functions and classes, in: Boffa, M., van Dalen, D., and McAloon, K. (eds.), *Logic Colloquium '78: Proceedings of the Logic Colloquium at Mons, 1978*, pp. 159-224, North-Holland, Amsterdam (1979).

Feferman, S. [1985] A theory of variable types, *Rev. Colombiana de Matemáticas*, XIX, 95-106.

Felty,A., and Miller, D. [1988] Specifying Theorem Provers in a Higher-Order Logic Programming Language, in: *Proceedings of the Ninth International Conference on Automated Deduction, Argonne Ill, May 1988*, pp. 61–80, Springer Lecture Notes in Computer Science **310**, Springer-Verlag, Berlin/ Heidelberg/ New York (1988).

Gabbay, D. M., and Reyle, U. [1984], N-Prolog: an extension of Prolog with hypothetical implication, Part I, *J. Logic Programming* **1** (1984) 319-355.

Gabbay, D. M., and Reyle, U. [1989], Computation with run time skolemisation (N-Prolog Part 3), to appear.

van Gelder, A. [1987], Efficient loop detection in Prolog using the tortoise-and-hare technique, *J. Logic Programming* **4** (23–32), 1987.

Girard, J. [1989] *Proofs and Types*, Cambridge University Press, New York (1989).

Hallnäs,L, and Schroeder-Heister, P., [1987] A proof-theoretic approach to logic programming, SICS R88005 Research Report ISSN 0283-3638, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden (1987).

Hayashi, S. [1988], *PX: A Computational Logic*, MIT Press, Cambridge, Mass. (1988).

Howe, D. [1988], Computational metatheory in Nuprl, in: *Proceedings of the Ninth International Conference on Automated Deduction, Argonne Ill, May 1988*, pp. 238–257, Springer Lecture Notes in Computer Science **310**, Springer-Verlag, Berlin/ Heidelberg/ New York (1988).

Huet,G. [1987] A uniform approach to type theory, INRIA Laboratory Research Report No. 795 (February 1988). An earlier version is more accessible:

Huet, G. [1986], Deduction and computation, in: *Fundamentals in Artificial Intelligence*, ed. by Bibel, W. and Jorrand, P., Lecture Notes in Computer Science **232**, Springer-Verlag, Berlin/ Heidelberg/ New York (1986).

Kleene, S. C. [1952], *Introduction to Metamathematics*, van Nostrand, Princeton, N. J. (1952).

Kleene, S. C. [1952a], Permutability of inferences in Gentzen's calculi LK and LJ, in: *Two Papers on the Predicate Calculus*, A.M.S. Memoirs **10** (1952), A. M. S., Providence, R. I.

Lifschitz, V. [1986], On the declarative semantics of logic programming with negation, in: *Proc. Workshop on Foundationsof Deductive Databases and Logic Programming, Washington, D.C., August, 1986.*

Lloyd, J. W. [1984], *Foundations of Logic Programming*, Springer-Verlag, Berlin/ Heidelberg/ New York (1984).

Manthey, R., and Bry, Francois [1988], SATCHMO: a theorem prover implemented in Prolog, in: *Proceedings of the Ninth International Conference on*

Automated Deduction, Argonne Ill, May 1988, pp. 415–434, Springer Lecture Notes in Computer Science **310**, Springer-Verlag, Berlin/ Heidelberg/ New York (1988).

Martin-Löf, P. [1984], *Intuitionistic Type Theory*, Bibliopolis, Naples (1984).

McCarthy, J. [1987], Generality in artificial intelligence, *Communications of the ACM*, vol. **30**, no. 12, December, 1987, pp. 1029–1035.

McCarty, L. T., [1984], Programming directly in a nonmonotonic logic: extended abstract, in: *Proceedings, AAAI Nonmonotonic Reasoning Workshop*, New Paltz, N.Y, Oct 1984, pp. 325–336.

McCarty, L. T., [1988] Clausal intuitionistic logic I. Fixed-point semantics, *J. Logic Programming* **5**, 1–33 (1988).

McCarty, L. T., [1988a] Clausal intuitionistic logic II. Tableau proof procedures, *J. Logic Programming* (to appear).

Miller, D., Nadathur, G., and Scedrov, A. [1987], Hereditary Harrop Formulas and Uniform Proofs Systems, in: *Proceedings of the Symposium on Logic in Computer Science, Cornell University, June 1987*, pp. 98–105, IEEE Computer Society Press, Washington, D.C. (1987).

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. [TA], Uniform Proofs as a Foundation for Logic Programming, to appear in *Annals of Pure and Applied Logic*, available as Report MS-CIS-89-36, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

Nadathur, G. [1987], *A Higher-Order Logic as the Basis for Logic Programming*, Ph. D. Thesis, University of Pennsylvania, 1987.

Paulson, L. C. [1986] Natural deduction as higher-order resolution, *J. Logic Programming* **3** 237-258.

Schütte, K. [1977], *Proof Theory*, Springer-Verlag, Berlin/Heidelberg/New York (1977).

Stickel, M. [1986] A Prolog technology theorem prover: implementation by an extended Prolog compiler, in: *Proc. 8th CADE*, Oxford, 1986, pp. 573-587, Lecture Notes in Computer Science **230**, Springer-Verlag, Berlin/ Heidelberg/ New York (1986). Longer version is to appear in *J. Automated Reasoning*.

Stickel, M. [1988] A Prolog technology theorem prover, in: *Proceedings of the Ninth International Conference on Automated Deduction, Argonne Ill, May 1988*, pp. 752–753, Springer Lecture Notes in Computer Science **310**, Springer-Verlag, Berlin/ Heidelberg/ New York (1988).

Thistlewaite, P.B., McRobbie, M.A., and Meyer, R.K., *Automated Theorem Proving in Non-Classical Logics*, Pitman, London, and Wiley, New York (1988).