An Intuitionistic Predicate Logic Theorem Prover

DAN SAHLIN, TORKEL FRANZÉN AND SEIF HARIDI, *SICS, Box 1263, S-164 28 Kista, Sweden. E-mail: {dan,torkel,seif}@sics.se*

Abstract

A complete theorem prover for intuitionistic predicate logic based on the cut-free sequent calculus is presented. It includes a treatment of 'quasi-free' identity based on a delay mechanism and a special form of unification. Several fairly far-reaching optimizations of the basic algorithm essential to its performance are introduced. The paper concludes with a set of benchmarks and execution data which may facilitate a comparison of algorithms for intuitionistic logic. The system is available in source code from SICS.

Keywords: automatic theorem proving, intuitionistic logic, equality theory.

1 Introduction

Theorem proving in intuitionistic predicate logic is not a highly developed subject, no doubt because it has as yet few applications. In the past few years, however, work has been pursued (mostly independently) here and there, particularly on intuitionistic propositional logic. In the present paper we present a complete theorem prover for intuitionistic predicate logic with (a restricted) identity, based on the rules of the intuitionistic sequent calculus, in the hope that the techniques, concepts, and examples presented will be of interest to others working in the field.

The calculus of sequents, formulated by G. Gentzen, is a system of rules for deriving expressions, called sequents, of the form

$$\Gamma \rightarrow \Delta$$

where Γ and Δ are sequences of predicate logic formulae. In the intuitionistic version of the calculus to be used here, Δ always consists of exactly one formula, and Γ is a finite sequence of formulae, possibly empty. Thus a sequent has the form

$$A_1, \ldots, A_n \rightarrow C$$

where $n \geq 0$. A sequent will here be interpreted as a predicate logic formula in an alternative notation: if $n = 0$ the sequent is interpreted as the formula C, and for $n > 0$ we read it as the formula $A_1 \& \ldots \& A_n \supset C$.

The idea of using the rules of the classical sequent calculus in automatic theorem proving is as old as the subject itself. In particular, the pioneering paper [5] contains a presentation of the basic ideas developed in this article. However, in the case of classical logic, the sequent calculus was soon displaced in automatic theorem proving by the much more efficient resolution method. To see why it makes sense to develop

J. Logic Computat., Vol. 2 No. 5, pp. 619–656 1992

and refine the method based on the sequent calculus in the case of intuitionistic logic, one must appreciate the great difference in formal properties between that logic and classical predicate logic, as reflected for example in the fact that there is no prenex form and monadic intuitionistic predicate logic is undecidable. For an overview of intuitionistic logic and its peculiarities, see [1].

Automatic theorem provers can be classified as incorporating heuristic or 'intelligent' methods to a greater or lesser extent. Since intuitionistic logic is computationally difficult, one would expect a practical intuitionistic theorem prover to draw on heuristic methods to a considerable extent. The algorithms to be presented here, however, are chiefly 'mechanical' in character. That is to say, with few exceptions, there is no logical or pattern-based analysis of the set of premisses, and no checking for special cases. Instead a set of rules is applied to each formula as it appears, without regard to the formulae that appeared before. The aim is to produce such a mechanical procedure which is not grossly inefficient, and yields a complete theorem prover for intuitionistic logic. Of course this is a losing battle. It would be idle to pretend that the present system is a satisfactory theorem prover, since it is easy to find fairly short examples that take 'forever' to prove. (See in particular the problematic formulae given in the appendix.) Nevertheless we consider the material in this paper to be of general interest, for several reasons:

1. A number of useful and non-obvious techniques are presented by which large classes of problems that are otherwise intractable (on a sequent calculus approach) become solvable. These techniques may well be applicable in other contexts. Also, the algorithm yields a reasonably efficient decision procedure for propositional intuitionistic logic (presented explicitly in Section 17).
2. A theorem prover is presented which is complete for problems involving free variables, i.e. all provable closed instances of a given formula are presented through backtracking. This feature (inspired of course by logic programming) is combined with a treatment of identity that allows us to pose questions and receive replies of a form not traditionally considered in theorem proving (and not available in standard logic programming). The identity theory, including a unification algorithm and a delay mechanism, is applicable in a classical as well as a constructive context.
3. The system is available in source code from SICS by anonymous ftp, and may be used as a starting point or for purposes of comparison by others with an interest in these matters. To this end, we have also included an appendix containing a number of predicate logic problems, with execution times.

In brief summary, the following is the content of this article. The logical system — intuitionistic predicate logic with a restricted 'quasi-free' identity — to which the theorem prover applies is presented in Section 2 and Section 4. Section 3 discusses in general outline the kind of procedure used and the problems connected with it. The following sections present various fundamental aspects of the system: the presentation of answers to queries (Section 5), the unification algorithm (Section 6), the delay mechanism (Section 7), the rotation of formulae in applications of rules (Section 8). A full specification of the basic algorithm is given in Section 9, while Section 10 completes the description of the top level procedure for answering questions. Sections 11–16 introduce various modifications of the basic procedure which form an important

part of the working system. Section 17 describes a decision procedure for propositional logic obtainable from the algorithm. The Appendix, finally, contains information on implementations together with a collection of problems and execution data.

As far as theory is concerned, the main result needed but not established in the present article is that the algorithm is in fact complete. Here only the soundness of the methods used will be considered. In the presentation of the algorithm in this article we have tried to avoid introducing unnecessary formalities. Our aim is to describe a fairly complex algorithm in clear terms, giving formal definitions when they are needed, and stating but not always proving the technical observations on which it rests.

2 The system GI without identity

Although some acquaintance with intuitionistic predicate logic will be presupposed in the following, an explanation of our notation and terminology is in order.

The language used will be that of predicate logic with equality. Two points should be noted: the use of the constant $\perp$ (interpreted as a logically false statement) as a logical primitive instead of negation, and the separation of variables into those that are only used as bound variables (called variables) and those that are only used as free variables (called parameters). The negation $\neg A$ is defined as an abbreviation of $(A \supset \perp)$. The letters x, y, z will be used for variables and α, β for parameters. Thus we have *terms* which are either parameters or individual constants or composite terms $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and f is an n-place function symbol. Individual constants will also be regarded as 0-place function symbols. The letters s, t, u, v will be used to denote terms. Note that variables do not occur in terms. The *formulae* are either atomic or composite. Atomic formulae are the special atomic formula $\perp$ (falsum or the absurdity) and $p(t_1 \ldots, t_n)$ where p is an n-place predicate symbol. We also admit 0-place predicate symbols, so that p is an atomic formula for every 0-place predicate symbol p. Composite formulae are $(A \supset B), (A \equiv B), (A \vee B), (A\&B), \forall x A(x/\alpha), \exists x A(x/\alpha)$. Here $A(x/\alpha)$ stands for the result of substituting x for every occurrence of α in A. When this notation is used it is presupposed that α does not occur in A within the scope of any quantifier $\forall x$ or $\exists x$. The expressions $A(t/\alpha)$ and $u(t/\alpha)$ are interpreted similarly. A *closed* term or formula is one not containing any parameters. Among the predicate symbols is the identity symbol $=$. Equalities are written in the usual way as $s = t$. The letters A, B, C, D will be used to denote formulae. Parentheses will normally be omitted in accordance with the following conventions: (1) the outermost parentheses are omitted when formulae occur in isolation; (2) association to the right is used for $\vee$ and &; (3) $\vee$ and & bind harder than $\supset$ and $\equiv$. Thus for example $A \vee B \supset C\&D\&E$ stands for $((A \vee B) \supset (C\&(D\&E)))$, whereas $A\&B \vee C$ is undefined.

It will be assumed that an unlimited supply of function symbols is available in the language.

The intuitionistic system defined by the system of rules given below will be referred to as GI. The system GF — taken as basic in this article — which differs from GI in including identity rules is presented in Section 4.

The composite formula introduced in the antecedent or consequent in an application of a rule is called the *principal* formula. To make the rules more readable, they are

formulated with the principal formula leftmost in the antecedent. The rules are to be understood, however, as covering every permutation of the formulae in the antecedent. For example, $\& \rightarrow$ covers every step of the form

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

where Γ is a permutation of A, B, Δ and Γ' is a permutation of $A\&B, \Delta$.

GI differs from standard formulations of the (cut-free) intuitionistic sequent calculus in two respects: the absence of a separate contraction rule and the inclusion of the rules for $\equiv$. Contraction will be commented on in Section 3. The reason for the inclusion of the $\equiv$-rule — theoretically unnecessary, since $A \equiv B$ is intuitionistically equivalent to $(A \supset B)\&(B \supset A)$ — is that this makes the proof procedure more efficient.

2.1 The system GI

Axioms: (called logical axioms)

$$\frac{}{\bot, \Gamma \rightarrow C} \qquad \frac{}{B, \Gamma \rightarrow B}$$

Rules

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

$$\frac{A, \Gamma \rightarrow C \quad B, \Gamma \rightarrow C}{A \vee B, \Gamma \rightarrow C} \vee \rightarrow \qquad \frac{\Gamma \rightarrow A_i \quad i \text{ is 1 or 2}}{\Gamma \rightarrow A_1 \vee A_2} \rightarrow \vee$$

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

$$\frac{A_1 \equiv A_2, \Gamma \rightarrow A_i \quad A_1, A_2, \Gamma \rightarrow C \quad i \text{ is 1 or 2}}{A_1 \equiv A_2, \Gamma \rightarrow C} \equiv \rightarrow \qquad \frac{A_1, \Gamma \rightarrow A_2 \quad A_2, \Gamma \rightarrow A_1}{\Gamma \rightarrow A_1 \equiv A_2} \rightarrow \equiv$$

$$\frac{A, \Gamma \rightarrow C}{\exists x A(x/\alpha), \Gamma \rightarrow C}\,(*)\exists \rightarrow \qquad \frac{\Gamma \rightarrow A(t/\alpha)}{\Gamma \rightarrow \exists x A(x/\alpha)} \rightarrow \exists$$

$$\frac{\forall x A(x/\alpha), A(t/\alpha), \Gamma \rightarrow C}{\forall x A(x/\alpha), \Gamma \rightarrow C} \forall \rightarrow \qquad \frac{\Gamma \rightarrow A}{\Gamma \rightarrow \forall x A(x/\alpha)}\,(*) \rightarrow \forall$$

(*) Restriction on the rules $\exists \rightarrow$ and $\rightarrow \forall$: the parameter α must not occur in C or in any formula in Γ.

Note: An equivalent system will be obtained if we restrict the logical axioms to sequents $B, \Gamma \rightarrow B$ where B is atomic.

3 The basic procedure and its problems

In seeking a proof of a sequent Σ we start with Σ and try to construct a proof of Σ from the bottom upwards. In fact we will consistently regard the rules in this light, so when an 'application' of a rule is spoken of in the following, what is intended is always a backwards application: a step from the conclusion to the premiss or premisses. The first problem we encounter in such a construction of a proof is how to deal with the substitutions required in applications of the rules $\forall \rightarrow$ and $\rightarrow \exists$. The solution is to introduce a meta-logical variable — called 'dummy variable' in [5] — to be used in substitutions. At various points in the procedure we will attempt to perform unifications with respect to these variables so as to produce an axiom. Since we use parameters instead of free variables in the logical language, we will simply speak of these meta-logical variables as 'free variables'. The letters X, Y, Z, W will be used as free variables. Expressions obtained by substituting free variables into formulae will be called *free-variable formulae*. *Free-variable terms* and *free-variable sequents* are defined similarly. We will occasionally drop the 'free-variable' when no confusion is possible. We also need a name for the structures with which the algorithm works, i.e. sequent calculus deductions using free-variable sequents. Rather than using the cumbersome 'free-variable deduction' we will speak of these structures as *attempted proofs*.

Unification with respect to free variables is complicated at least by the need for an occur check and a mechanism for ensuring that the restrictions on the rules $\exists \rightarrow$ and $\rightarrow \forall$ are respected. The kind of unification to be used in the present system is in addition made considerably more complicated by the treatment of identity. The unification algorithm is presented in Section 6.

The other major problems connected with the basic algorithm are those of contraction, non-determinism, and what will here be called 'rotation'. Before we describe in outline the treatment of these problems, the relevant aspects of the rules call for some comments. First there is the concept of invertibility. That the rules of the sequent calculus are logically valid means that the conclusion in an application of a rule is logically valid if the same is true of the premiss or premisses. A rule is called *invertible* if the converse holds, i.e. if the logical validity of the conclusion implies the logical validity of the premisses. There are four non-invertible rules in GI: $\rightarrow \exists, \rightarrow \vee, \supset\rightarrow, \equiv\rightarrow$. (That these and only these rules are non-invertible is semantically obvious.) The rules $\supset\rightarrow$ and $\equiv \rightarrow$ are *semi-invertible*: the right premiss of these rules is valid whenever the conclusion is, but this does not hold for the left premiss.

Invertibility has large consequences for the bottom-up construction of derivations. Suppose we are seeking to formulate a complete system of rules for the sequent calculus, i.e. a system such that every logically valid sequent is provable. If the system is complete it will always be possible to construct a proof of a valid sequent by means of some kind of bottom-up procedure. If a rule is invertible we know that the task of proving the conclusion can always be reduced to that of proving the premisses. If, on the other hand, a rule is not invertible, the corresponding reduction may lead to an impossible task. As a consequence, an invertible rule is always easier to deal with than a non-invertible rule. An invertible rule may be applied deterministically at any point in the (bottom-up) procedure, that is, we need not consider alternatives to applying that particular rule, since we know that the premisses will still be provable if the original sequent is provable. If we apply a non-invertible rule to a provable

sequent, however, there is no guarantee that the premisses can be proved, and we must be prepared to backtrack and try another rule or try several rules in parallel.

A semi-invertible rule admits a certain amount of determinism: if the rule is invertible w.r.t. the right premiss and the right premiss turns out not to be provable, we need not consider any alternative to applying the rule in question, but can reject the conclusion as unprovable. If, on the other hand, the right premiss is proved but the left premiss not, we must still be prepared to try applying another rule or using another formula.

The second aspect of the rules that calls for some comment is contraction. The sequent calculus was originally formulated with a *contraction rule* of the form

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

When proofs are constructed in bottom-up fashion, use of the contraction rule means that a copy is made of the principal formula before it is 'dissolved' (as it appears when the rules are applied backwards) so that essential information will not be lost. The rules given above do not include any contraction rule. Instead the unavoidable uses of contraction have been incorporated into the formulation of the rules $\supset\rightarrow$, $\equiv\rightarrow$ and $\forall\rightarrow$.

Contraction is responsible for the lack of any computable bound on the size of a proof of a given sequent. If no use is made of contraction — i.e. if no contraction rule is used, and the rules $\supset\rightarrow$, $\equiv\rightarrow$, and $\forall\rightarrow$ are formulated without any repetition of the principal formula — a logical system results which is a decidable fragment of standard (classical or intuitionistic) logic.[1]

Hence the contraction problem: the use of the rules $\supset\rightarrow$, $\equiv\rightarrow$, and $\forall\rightarrow$ may never end because of contraction. Of course, in general we must expect any proof procedure to go on forever if a sequent for which a proof is sought is not in fact valid. The problem here, however, is that even if the sequent is valid we may get trapped in unending sequences of applications of rules which cannot lead to a proof. This is because of the non-invertibility of the rules $\rightarrow\exists$, $\rightarrow\vee$, $\supset\rightarrow$, $\equiv\rightarrow$. (In seeking proofs in the classical sequent calculus, in contrast, it is possible, because every rule is invertible, to avoid all such traps by systematically using each formula in turn.)

One method of dealing with the contraction problem is to use a parallel or breadth-first procedure in seeking proofs. Here, however, we will deal only with sequential or depth-first procedures. Thus we must impose a limit on the use of contraction. A *contraction parameter* is associated with each universal formula, implication, and equivalence. The value of the contraction parameter determines how many times the formula may be used in any one branch of the proof being sought. When the contraction parameter falls to 0, the rules $\forall\rightarrow$, $\supset\rightarrow$, and $\equiv\rightarrow$ can only be applied *without* repetition of that formula as principal formula. If the attempt to find a proof using one set of values for the contraction parameters fails, we increase those parameters and try again. (This kind of method is also adopted in [3].)

This will be recognized as a variant of the familiar 'bounded depth first iterative deepening'. The details, indeed, are different, since the contraction parameters

[1] For a study of this fragment in the intuitionistic case, see [2]. In intuitionistic *propositional* logic, there is a computable upper bound on the use of contraction since that logic is decidable. (See Section 17.) In predicate logic, however, there is no such bound either in the case of implications and equivalences or in that of universal formulae.

considered here are a finer instrument than the wholesale depth parameter of that technique. The problems of using this kind of method are the same, however: how are we to choose (1) the initial values of the contraction parameters, and (2) the value by which those parameters are incremented on failure? Since these two choices often make a dramatic difference to execution times, it would be nice to be able to incorporate some intelligent way of making them into the algorithm. We know on general grounds, however, that making intelligent choices of the kind (1) and (2) is a problem of unlimited difficulty, much like theorem proving itself. In the system presented here we have not attempted to include any such choices, but simply leave it to the user to specify the values of the parameters involved.

The problem of non-determinism consists of course in the fact that we must, as noted above, be prepared to apply the non-invertible rules in every possible order with every possible choice of principal formula. This non-determinism leads to a combinatorial explosion already at the propositional level, and with the introduction of variables and quantifiers it becomes totally crippling. Hence a number of techniques will be introduced below to reduce the amount of non-determinism: compaction, consequent locking, implication locking, sifting of bindings, use checking. The obvious (and traditional) first step towards reducing the non-determinism is to stipulate that a non-invertible rule will be applied only when no invertible rule is applicable, and that the semi-invertibility of the rules $\supset\!\rightarrow$ and $\equiv\!\rightarrow$ will be exploited as described above.[2] (As it turns out, semi-invertibility will only be partially exploited in the present algorithm — see Section 14.)

The rotation problem is less straightforward: it concerns the question in what order we are to use the formulae in a sequent (as principal formula in an application of a rule). To obtain a complete proof procedure we must ensure that every formula is used sooner or later, but there are many ways of doing this, some of which are worse than others.

We will speak of *horizontal* and *vertical* rotation. Horizontal rotation concerns the use of formulae in a sequent at one and the same level of an attempted proof. That is, we have the task of proving a sequent $A_1, \ldots, A_n \rightarrow C$, and must be prepared to try using each of the formulae $A_1, \ldots, A_n, C$ as principal formula in an application of a rule. To stipulate a horizontal rotation scheme is to say in what order we are to use $A_1, \ldots, A_n, C$. Vertical rotation concerns the universal formulae, implications, and equivalences, which are carried over into the premiss sequents: what is to be their relative order in the premisses? For example, should we keep using the same universal formula in a branch of an attempted proof as long as any contractions remain, or should we try another universal formula?

One horizontal rotation principle is immediately dictated by the need to reduce non-determinism, as noted in the preceding paragraph. We have, therefore, two questions regarding horizontal rotation. First, in what order are we to use the formulae that may appear as principal formula in an application of an invertible rule, i.e. conjunctions, disjunctions, and existential formulae in the antecedent, and conjunctions, universal formulae, implications, and equivalences in the consequent? Here we have found no basis for specifying any particular order (with one minor exception), so these rules will be simply be applied in whatever order the formulae happen to arise. The

[2]The invocation of semi-invertibility is of course a bit more complicated when bounded contraction is used, since failure to prove a formula at a certain contraction depth does not in general imply that the formula is unprovable.

second question concerns the order in which to use formulae as principal formula in an application of a non-invertible rule. Certain specific choices in this regard are dictated by the optimizations mentioned above in connection with the problem of non-determinism. In addition we have implemented a principle of using universal formulae before implications and equivalences.

As regards vertical rotation, we treat universal formulae and implications (or equivalences) differently. Universal formulae follow a principle of strict vertical rotation, whereas implications and equivalences are carried over in unchanged order to the premisses. These principles and their justification, such as it is, will be presented in Section 8.

4 Identity: the system GF

In order to improve the prospects for a computationally feasible identity theory, we will use the free interpretation of terms. That is, the universe is assumed to be freely generated from some set by the operations for which function symbols occur in the language. In fact, usually we take the universe of discourse to be the Herbrand universe of closed terms. To obtain a complete system of logic we must however consider a wider class of interpretations, as will be commented on later in this section.

The resulting system, which will be called GF, has the following rules and axioms for identity in addition to the rules and axioms of GI:

Axioms:

Equality axiom:

$$\Gamma \rightarrow t = t$$

Inequality axioms:

$$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m), \Gamma \rightarrow A \quad m, n \geq 0$$

for different function symbols f, g

$$\alpha = s[\alpha], \Gamma \rightarrow A$$

$$s[\alpha] = \alpha, \Gamma \rightarrow A$$

In the last two axioms, $s[\alpha]$ is a term which properly contains the parameter α.

Rules:

$$\frac{s_1 = t_1, \ldots, s_n = t_n, \Gamma \rightarrow A}{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), \Gamma \rightarrow A} \text{ injectivity rule}$$

$$\frac{\Gamma(s/\alpha) \rightarrow A(s/\alpha)}{a = s, \Gamma \rightarrow A} \text{ replacement rule}$$

$$\frac{\Gamma(s/\alpha) \rightarrow A(s/\alpha)}{s = \alpha, \Gamma \rightarrow A} \text{ replacement rule}$$

In the replacement rules, s is a term which does not contain the parameter α. The injectivity rule has a special case $n = 0$ in which an equation $e = e$, where e is an individual constant, is introduced in the antecedent. (Corresponding to the special case of the replacement rules in which s is α.)

The inequality axioms and the injectivity rule are not valid on the ordinary interpretation of identity, unlike the replacement rules. A computationally pleasant aspect of the identity rules is that none of them incorporates contraction: thus formulae $s = t$ are eliminated once and for all when a proof is sought. That no contraction is needed is clear since these three rules are all invertible (in the case of the replacement rules because of the restriction that α does not occur in s). Nevertheless we will find that it is sometimes necessary in seeking a proof to retain part of the information contained in the premiss $\alpha = s$ or $s = \alpha$ in an application of a replacement rule.

Since being freely generated (in the standard algebraic sense) from some set is not in fact a first-order property of structures, the above axioms and rules necessarily hold for a wider class of structures, which we will call *quasi-free*. Thus a structure is quasi-free if the functions are injective and have pairwise disjoint ranges, and no sequence of applications of functions can lead from an individual a to a.

To amplify this point: note that the axioms of GF do not rule out, for example, an infinitely descending sequence $a_0, a_1, \ldots$, such that $f(a_{i+1}) = a_i$ for all i. No matter what rules and axioms valid in all freely generated structures we put down, they will have models containing such sequences. Thus we cannot formulate a logic of free identity in the sense of a first-order logic all of whose models are freely generated. We could of course extend the present system to include, for example, induction principles valid in all freely generated structures. We know, however, that the set of formulae valid in all freely generated structures is not effectively enumerable, so any formalizable extension of GF will necessarily be incomplete regarded as an identity theory for such structures. Hence we opt instead for a complete logical theory of quasi-free identity in general structures.

One particular possibility of extending the identity axioms of GF calls for special comment. Identity is not in general a decidable relation in intuitionistic logic. That is, the formula $\forall x \forall y(x = y \vee \neg x = y)$ is not provable. Identity between the elements in a Herbrand domain generated by a decidable set of function symbols and individual constants *is* however intuitionistically decidable (as is easily proved by induction). Thus if we intend GF to be a logical theory of Herbrand domains only, rather than of quasi-free structures, it is logically proper to include $\forall x \forall y(x = y \vee \neg x = y)$ as an axiom or a rule. Since we haven't found any way of incorporating the decidability of identity that greatly improves on the mere addition of $\forall x \forall y(x = y \vee \neg x = y)$ to the antecedent of a sequent, we have not included decidability of identity in the system.

We have spoken of structures and logical validity above without benefit of any formal definitions. Definitions of these concepts and a completeness proof for GF as a formalization of the logic of quasi-free Kripke models can be found in [4]. The completeness of GF implies in particular that all the usual identity rules are derivable in GF, and that the cut rule holds as a derived rule; i.e. if $\Gamma \rightarrow A$ and $A, \Delta \rightarrow B$ are provable in GF, then so is $\Gamma, \Delta \rightarrow B$.

Some remarks about the interest and utility of the identity theory of GF may be in order. Clearly we cannot deal with, for example, such standard problems for identity logic theorem provers as elementary theorems in group theory, since no operations

are associative in the sense of quasi-free identity: $f(\alpha_0, f(\alpha_1, \alpha_2)) = f(f(\alpha_0, \alpha_1), \alpha_2)$ implies $\alpha_0 = f(\alpha_0, \alpha_1)$ by the injectivity rule, and this conclusion is false by the inequality axioms. To appreciate the range of application of quasi-free identity we must turn from mathematics to the kind of applications found in database handling and logic programming.

5 Queries, answers and completeness

As stated in Section 1, our procedure for proving theorems in GF applies to free-variable formulae. For convenience in describing the system we will take formulae rather than sequents as input — in practice it is of course a simple matter to allow sequents. We will adopt the terminology of logic programming and refer to a free-variable formula fed to the system as a *question*. The free variables in a question $A(X_1, \ldots X_n)$ will be called *input variables*.

The question $A(X_1, \ldots, X_n)$ gives the automatic proof system the task of finding closed terms $t_1, \ldots, t_n$ such that the formula $A(t_1, \ldots, t_n)$ is provable in GF. Thus as is usually the case, we do not demand the production of a proof in GF, but only a statement of provability. Since there will in general be infinitely many such sequences $t_1, \ldots, t_n$, not all of which can be presented or found together, the system must, to be a complete theorem prover, have a mechanism for presenting a possibly infinite sequence of answers to a question. This again is familiar from logic programming. The answers given by the system will consist of *bindings* and *constraints*. These terms will be formally defined below; but first we present an example. The following question is put to the system

$$p(a)\&\exists w \exists y \forall z(p(z) \supset z = w \vee z = y) \supset \neg(p(x_1)\&p(x_2)\&p(x_3))$$

The premiss in this example formalizes 'Jane is a philosopher, and there are at most two philosophers'. We ask for terms t_1, t_2, t_3 such that the premiss implies that not all of t_1, t_2, t_3 are philosophers.

Among the answers given there will be four essentially different solutions:

1. $X_1 = A, X_2 = B, X_3 = C$ where $A \neq B$ and $A \neq C$ and $B \neq C$;
2. $X_1 = A, X_2 = B$ where $A \neq B$ and $A \neq a$ and $B \neq a$;
3. $X_1 = A, X_3 = C$ where $A \neq C$ and $A \neq a$ and $C \neq a$;
4. $X_2 = B, X_3 = C$ where $B \neq C$ and $B \neq a$ and $C \neq a$.

In ordinary language: the implication is valid if t_1, t_2, t_3 are three different closed terms, or if two of them are different from a and from each other. That these solutions are essentially different means that none of them subsumes any of the others. The definition of completeness stated below does not say anything about redundancies in the presentation of solutions. In practice it is all but necessary to check for subsumption, since otherwise we will be flooded with seemingly interminable repetitions of solutions.

Now for some general definitions:

- An affirmative *answer* — which we also call a *solution* — delivered by the system has two parts: a possibly empty set of *bindings* and a possibly empty set of *constraints*.

- A binding has the form $X_i = S$, where X_i is an input variable and S a free-variable term in which no input variable occurs. The free variables that occur in S will be called *auxiliary* variables. No input variable occurs in more than one binding in the set.
- The stipulation that no input variable occurs in S is convenient in presenting solutions and entails no restriction on bindings. For example, we do not present a set of bindings in the form $X_1 = X_2, X_3 = g(X_1)$, but instead as $X_1 = Y, X_2 = Y, X_3 = g(Y)$.
- The meaning of the binding regarded as an answer is that the input variable X_i may be taken to stand for any closed term obtainable by substituting closed terms for the auxiliary variables in S (those closed terms possibly being subject to further conditions stated in the constraints).
- A constraint has the form $Y \neq T$, where Y is an auxiliary variable and T is an auxiliary-variable term with place markers in which Y does not occur. By this we mean an expression formed from a term which in addition to auxiliary variables (i.e. free variables in bindings other than input variables) may contain *place markers* of the form $\%n$. (T is not allowed to *be* a place marker, however.) Y can occur as the left term in any number of constraints.
- The meaning of the constraint $Y \neq T$ is that Y is *not* a term obtainable from T by substituting closed terms for the place markers and the values of the auxiliary variables for those variables. Again it is a matter of convenience to stipulate that no input variables occur in constraints.
- In slightly more formal terms: a sequence $t_1, \ldots, t_n$ of closed terms is an *instance* of a set of bindings and constraints if there are substitutions of closed terms for the auxiliary variables such that the equations $X_i = S$ and inequations $Y \neq T$ with t_j substituted for X_j everywhere ($j = 1, \ldots, n$) are true for all values (in the domain of closed terms) of the place markers.
- Besides affirmative answers, the system can produce negative answers, a simple 'no' meaning that no solutions, or no further solutions, can be found. In general, of course, the search for solutions will go on for ever.
- Using this terminology, we can define what it means for the proof system to be sound and complete: *soundness* means that for every answer to a question $A(X_1, \ldots, X_n)$ delivered by the system and every instance $t_1, \ldots, t_n$ of that answer, $A(t_1, \ldots, t_n)$ is provable in GF. *Completeness* means that for every sequence $t_1, \ldots, t_n$ of closed terms, if $A(t_1, \ldots, t_n)$ is provable in GF, then there is some answer delivered by the system to the question $A(X_1, \ldots, X_n)$ such that $t_1, \ldots, t_n$ is an instance of that answer.

Another example illustrates the role of the place markers in constraints. We put the question

$$\forall x(p(x) \equiv \exists y \exists z(p(y) \& p(z) \& x = f(y, y, z))) \supset \neg p(W)$$

Here we have an infinite sequence of essentially different solutions:

1. $W = A$ where $A \neq f(\%1, \%1, \%2)$
2. $W = f(A1, A1, A2)$ where $A1 \neq f(\%1, \%1, \%2)$

3. $W = f(A1, A1, A2)$ where $A2 \neq f(\%1, \%1, \%2)$
4. $W = f(f(A1, A2, A2), f(B1, B1, B2))$ where $A1 \neq f(\%1, \%1, \%2)$

....

An affirmative answer is of little interest (although by definition correct) if it does not have any instances. In fact this cannot happen given the definition of an answer and the unlimited supply of function symbols in the language. We formulate this observation as a

CONSTRAINT LEMMA
Every set of bindings and constraints has at least one instance.

PROOF. We argue by induction on the number of auxiliary variables in the set. If there are no auxiliary variables, there are no constraints, but only a set of bindings of input variables to closed terms. Now suppose the auxiliary variable Y occurs in the constraints $Y \neq T_1, \ldots, Y \neq T_n$, where $T_1, \ldots, T_n$ are not variables, and also in the constraints $Y \neq W_1, \ldots, Y \neq W_k$, where the $W_1, \ldots, W_k$ are variables. Let a be a new individual constant (i.e. one that doesn't occur in any of the terms involved). Then a is different from $T_1, \ldots, T_n$ for every value of the place markers and the auxiliary variables in $T_1, \ldots, T_n$. Form a new set of bindings and constraints by (i) removing $Y \neq T_1, \ldots, Y \neq T_n$ and $Y \neq W_1, \ldots, Y \neq W_k$, (ii) substituting a for Y everywhere in the remaining bindings and constraints, (iii) adding the constraints $W_1 \neq a, \ldots, W_k \neq a$. An instance of this set together with the term a yields an instance of the original set. ∎

A variant of the constraint lemma will be used in Section 7 to justify the treatment of delayed sequents.

6 Variables, parameters and unification

In applying the rules $\exists \rightarrow$ and $\rightarrow \forall$ we introduce new parameters, and in applications of the rules $\rightarrow \exists$ and $\forall \rightarrow$ we introduce new free variables. We must ensure that the parameter restrictions on the rules $\exists \rightarrow$ and $\rightarrow \forall$ are respected, not only when these rules are applied, but when substitutions are made for the free variables.

To this end, we use the following method. Each sequent is annotated with a *parameter index* i, which is set to 0 in the bottom sequent. When one of the rules $\exists \rightarrow$ and $\rightarrow \forall$ is applied, we use the parameter α_i where i is the current parameter index and increment the parameter index of the premiss sequent to $i + 1$. In applications of other rules the index is unchanged. When we apply $\rightarrow \exists$ or $\forall \rightarrow$ we use a new variable — one that has not been used previously in the attempted proof — and furthermore annotate this variable with the current parameter index. We will write a variable X thus annotated as X^i. The meaning of the annotation is:

terms substituted for X^i must not contain any parameter α_k where $k \geq i$.

This method has the considerable advantage of making a single numerical parameter carry the information what is an admissible substitution for a variable. There is a difference, it will be noted, between the way variables and parameters are generated. Free variables are global: substitutions for a variable are carried out throughout an attempted proof, wherever the variable appears. Hence it is essential to make the new

variable X unique: it must not already appear elsewhere in the proof. Parameters, on the other hand, are not unique: the parameter α_i may already have appeared in other places when we introduce it in an application of $\exists \rightarrow$ or $\rightarrow \forall$. This is of no consequence, since we know that α_i does not occur in the present sequent.

A term t will be called *i-admissible* if it does not contain any parameter α_k for $k \geq i$. Thus a variable X^i varies only over i-admissible terms. A substitution of a term t for a variable X^i will be called *legal* if t is i-admissible.

In order to find free-variable terms to substitute for the variables we perform a unification: at certain points in the procedure — exactly where will be considered later — when we have a sequent $A, \Gamma \rightarrow B$, with A and B free-variable formulae, we try to unify A and B so as to get an axiom, thus introducing bindings of the free variables. Similarly we use unification of free-variable terms S and T to make an axiom out of a sequent $\Gamma \rightarrow S = T$. In the unification we must take into account the restrictions on the variables.

To see how this unification works, let us for the moment disregard everything having to do with identity and suppose that we are looking for a proof of a sequent in which the symbol $=$ does not occur.

The unification procedure applies to a pair of free-variable expressions (terms or formulae) and yields as result either a failure report or a set of bindings of free variables to free-variable terms. It is only the unification of a variable with a free-variable term that needs to be defined here: unification between non-variables proceeds precisely as in standard syntactic unification.

To unify the variable X^i with the free-variable term T, we first check whether X^i properly occurs in T. If it does, the unification fails. In the trivial case where T is the variable X^i itself, we return the empty binding. If X^i does not occur in T, we go on to check whether T is i-admissible. If it is not, the unification fails. If T is i-admissible, we give the binding of X^i to T as output, but in addition we adjust the restriction values of the variables in T: every variable Y^k in T for which $k > i$ is given the new restriction i and thus becomes Y^i. This will be called *i-adjusting* the term T. To i-adjust T is clearly necessary if the restriction on X^i is to remain in force after the substitution of T for X^i. The adjustment applies only as long as the binding of X^i to T holds: when that binding is undone in the course of the execution of a backtracking algorithm, the former restrictions on the variables in T must be restored.

To avoid any possibility of notational confusion, we will use the notation $X^i := T$ to denote the binding of X^i to T.

A few examples will illustrate these points. In these examples, we ignore contraction and everything not directly pertaining to the use of unification.

In trying to prove the sequent

$$\exists x \forall y p(x, y) \rightarrow \forall y \exists x p(x, y)$$

we will come to the free-variable sequent

$$p(\alpha_0, X^2) \rightarrow p(Y^2, \alpha_1).$$

Unifying the free-variable formulae succeeds with the bindings $X^2 := \alpha_1, Y^2 := \alpha_0$, and the corresponding substitutions will yield the usual proof of this sequent. If, on

the other hand, we try to prove

$$\forall x \exists y p(x, y) \rightarrow \exists y \forall x p(x, y)$$

we will come to

$$p(X^0, \alpha_0) \rightarrow p(\alpha_1, Y^0)$$

or a variant. In any case, the unification will fail because of the variable restrictions.

The need for parameter adjusting free-variable terms is shown, for example, by the formula

$$\exists x \exists y \forall z \exists w((p(x) \supset p(f(y, w))) \& (p(f(y, w)) \supset p(f(a, z)))).$$

This formula is not valid. In trying to prove it we will come to something like

$$(p(X^0) \supset p(f(Y^0, Z^1))) \& (p(f(Y^0, Z^1)) \supset p(f(a, \alpha_0))).$$

Here the left formula will be proved using the binding $X^0 := f(Y^0, Z^1)$. If we now go on to prove the right formula without adjusting the restriction on Z, we will succeed in proving it with the bindings $Y^0 := a, Z^1 := \alpha_0$, and thus the procedure will not be sound.

When seeking proofs of sequents in which identity does not occur the above unification is sufficient. When we take identity into account, a new kind of situation arises, one in which the unification described so far is insufficient. An example will make this clear. Suppose we wish to prove the following sequent (obtained through applications of quantifier rules which are here left out):

$$\alpha_1 = f(\alpha_2, \alpha_7), p(Y^6) \rightarrow p(f(X^6, \alpha_7)).$$

The only rule of GF applicable to this sequent is the left replacement rule, and applying that rule merely removes the first formula in the antecedent, since there are no substitutions to make. Nor is it possible to unify $p(Y^6)$ and $p(f(X^6, \alpha_7))$ in the sense of the procedure defined above, and indeed there is no substitution for the free variables which makes an attempted proof with this sequent as an end sequent into a proof. Hence we are stuck if there is nothing else we can do with the free variables.

There is something else to do, however: we can make legal substitutions for the free variables so as to produce a sequent that can be proved using replacement. By substituting α_1 for Y^6 and α_2 for X^6 we get

$$\alpha_1 = f(\alpha_2, \alpha_7), p(\alpha_1) \rightarrow p(f(\alpha_2, \alpha_7))$$

which is provable by one application of the replacement rule. Note that it would not have been possible to find this provable instance of the sequent if we had thrown away the first antecedent formula.

In general there may well be several different ways of making substitutions of this kind. For example, if our original sequent is

$$\alpha_1 = f(\alpha_2, \alpha_7), \alpha_3 = f(h(\alpha_6), \alpha_7), \alpha_4 = h(\alpha_6), p(Y^6) \rightarrow p(f(X^6, \alpha_7))$$

we have two possibilities. One is the substitution given above, the second possibility is to substitute α_3 for Y^6 and α_4 for X^6 getting

$$\alpha_1 = f(\alpha_2, \alpha_7), \alpha_3 = f(h(\alpha_6), \alpha_7), \alpha_4 = h(\alpha_6), p(\alpha_3) \rightarrow p(f(\alpha_4, \alpha_7))$$

which is provable by two applications of replacement.

In order to incorporate the detection of such substitution possibilities into the algorithm we have chosen to modify the unification procedure so that it is sometimes possible to unify X_i and T even though T is not i-admissible. This means that the corresponding substitutions in an attempted proof will not always yield a correct proof, since the parameter restrictions are violated. Instead we must understand the bindings produced by unification as follows: from those bindings together with a set of equations of the form $\alpha_j = S$ can be extracted substitutions and finishing applications of replacement rules by which an attempted proof is turned into a proof.

We can now give a full specification of the unification algorithm to be used. Two free-variable expressions S and T (terms or formulae) are unified relative to a sequence Uf of equations of the form $\alpha_j = V$ where V is a composite free-variable term. For each such equation in Uf, the following three conditions hold:

(Uf1) α_j does not occur in S, T, or V, or in any other equation in Uf;
(Uf2) V contains α_m for some $m > j$;
(Uf3) V does not contain any variable X^i where $i > j$.

The information carried by the equation $\alpha_j = V$ is that the term V is allowable as a value for X^i where $i > j$, even though V may not be i-admissible. How the sequence Uf is generated will appear from the full specification of the algorithm in Section 9.

To describe the procedure for unifying two expressions relative to Uf we again consider only the unification of X^i with T, the other cases agreeing with ordinary syntactic unification. Because of the different possibilities of substitution exemplified above, unification is no longer deterministic — there is not always a most general unifier. Hence there will be a finite number of choice points in the procedure, alternatives that will be systematically tried in the proof algorithm. These points are marked by the word 'alternative' in the description. The different outputs are obtained by making different choices at these points. Each output consists of a list of unification bindings $Y_j := S_j$ $(j = 1, \ldots, k)$ where S_j is a free-variable term. Corresponding to each such list is a substitution operation on free-variable expressions whereby S_j is substituted for every occurrence of Y_j $(j = 1, \ldots, k)$ until no occurrences of the Y_j remain. There are no circular unification bindings, so the order of substitution is immaterial.

To unify X^i with T we go through the following steps:

- If T is X^i, return the empty binding. If T properly contains X^i, return failure.
- Otherwise, check whether T is i-admissible. If so, i-adjust T and return the binding $X^i := T$.
- If T is not i-admissible, first check whether T is a parameter. If so, return failure. Otherwise pick an equation $\alpha_j = V$ in Uf where $j < i$ and try to unify T and V. If this succeeds, return the resulting bindings together with the binding $X^i := T$, after i-adjusting T.

  Each output from such a unification of T with the right side of some equation in Uf provides an alternative output for the unification of X^i and T.

As an alternative to the above step:

- T must be a composite term, say $f(T_1, \ldots, T_n)$. Let $T_{k_1} \ldots, T_{k_m}$ be the terms among $T_1, \ldots, T_n$ that are not i-admissible. Pick an equation $\alpha_j = V$ in Uf where

$j < i$ and try to unify T_{k_1} and V. Apply the substitution yielded by the resulting list of bindings to the remaining terms $T_{k_2}, \ldots, T_{k_m}$, obtaining $T'_{k_2}, \ldots, T'_{k_m}$ and continue in the same way with these terms. Return the union of the resulting lists of bindings together with the binding $X^i := T$, after i-adjusting T. Each combination of bindings from such a set of unifications of the non-i-admissible T_{k_j} with right side terms of equations in Uf provides an alternative output for the unification of X^i and T. If there is no such combination of bindings, the unification of X^i and T fails.

We must verify that a consistent set of bindings is output. This follows from the fact that X^i does not occur in T by the first step of the procedure, and does not occur in any $\alpha_j = V$ in Uf where $j < i$ by the condition Uf3. Hence no binding of X^i is returned by the other unifications performed.

In connection with the alternatives in the procedure, it should be recalled that the parameter adjustments of terms entailed by the introduction of one set of bindings must be undone when alternatives to those bindings are sought.

If the sequence Uf is empty, the unification procedure coincides with the simpler procedure not involving identity described earlier.

The justification for this unification procedure consists in the observation that if free-variable formulae A and B are unifiable relative to Uf, the sequent

$$\text{Uf}, A \rightarrow B$$

is provable using replacement, once suitable legal substitutions, found in the unification procedure, have been made for the free variables. (Similarly for sequents with consequent $S = T$.) The details can be verified by an inductive argument.

One important special case should be noted: a simple inspection of the procedure shows that a variable X^0 with restriction parameter 0 cannot be unified with a T containing parameters. This observation is needed to verify that bindings of input variables yield provable instances of a question.

7 The delay mechanism. Producing answers

The inequality axioms of GF make it possible to introduce a delay mechanism which applies to two classes of free-variable sequents.

The first kind of delay, called a *parameter delay*, occurs when we encounter a sequent of the form

$$\alpha_i = S, \Gamma \rightarrow C \quad \text{or} \quad S = \alpha_i, \Gamma \rightarrow C$$

where S is a free-variable term not containing α_i in which occurs at least one variable X^k with $k > i$.

If this holds, S will be said to satisfy the *i-parameter condition.*

When a parameter delay is triggered it will be because of the detection of some particular variable X^k in S with $k > i$: this variable will be called the *delaying variable.* Thus a parameter delay consists of a delayed sequent together with a delaying variable.

The idea in delaying a sequent of the above form is that we may not have to do anything more about proving it, for if the variable X^k remains unbound it is always possible to make the sequent into an inequality axiom by substituting a term

containing α_i for X^k Should S contain α_i, we already have an inequality axiom, whatever terms are substituted for the variables.

The second kind of delay, called a *variable delay*, is triggered by sequents of the form

$$X^i = S, \Gamma \rightarrow C \quad \text{or} \quad S = X^i, \Gamma \rightarrow C$$

where S is a free-variable term other than a parameter, i.e. a composite free-variable term or a free variable, in which X^i does not occur. In this case too there will in general be several possible choices of delaying variable, corresponding to different possibilities of making the sequent into an inequality axiom. X^i can always be taken as delaying variable, but in addition any variable S with a restriction index that is greater than or equal to the restriction index of every other variable in $X^i = S$ can be designated as delaying variable. In case S is also a variable, say Y^j, we will always follow this second principle and take the delaying variable to be the variable with the greater restriction index (i.e. the less restricted variable). Variable delays, like parameter delays, will be taken to consist of the delayed sequent together with a delaying variable.

In talking about delays, we will sometimes refer to a delay as simply 'the delay $X^i \neq S$' or 'the delay $\alpha_i \neq S$', indicating only the formula prompting the delay, when the remainder of the sequent is not germane to the point at issue.

A delayed sequent remains in cold storage as long as no binding is created by which the condition for delaying the sequent is destroyed. When such bindings do arise, the delayed sequent is awakened and input anew to the proof procedure. Thus, if there are delayed sequents, a watch must be kept upon the delaying variables in those sequents. A full description will be given in Section 9.

When a sequent has been proved, the contraints, if any, given as part of the answer will be constructed from the delayed sequents. Not every delayed sequent will be relevant to the answer, however. In particular, if there are no input variables in the question, all remaining delayed sequents are ignored when a proof of the input formula has been found. The justification for this and the details of the construction of answers from bindings and delayed sequents are the topic of this section.

DELAY LEMMA

For any set of delays, there is a legal substitution of terms for the delaying variables such that whatever further substitutions of terms are made for variables other than delaying variables, the delayed sequents become inequality axioms.

PROOF. We argue by induction on the number of delays. Pick a delaying variable X. Suppose X is delaying variable in

1. the variable delays $X \neq S_1, \ldots X \neq S_n$ where $S_1, \ldots, S_n$ are composite terms;
2. the variable delays $X \neq W_1, \ldots, X \neq W_m$ where $W_1, \ldots, W_m$ are variables;
3. the variable delays $Y_1 \neq T_1, \ldots, Y_k \neq T_k$ (where X occurs in T_i for $i = 1, \ldots k$);
4. the parameter delays $\alpha_{i_1} \neq U_1, \ldots, \alpha_{i_p} \neq U_p$.

By the induction hypothesis, there is a substitution for the other delaying variables such that all other delays become inequality axioms whatever the value of X. Let $W_1^*, \ldots, W_m^*$ be the terms substituted for $W_1, \ldots, W_m$ in that substitution. (If W_i is not a delaying variable, W_i^* is W_i.) Similarly $Y_1^*, \ldots, Y_k^*$ are the terms substituted for

$Y_1, \ldots, Y_k$. Now substitute $f(\alpha_{i_1}, \ldots, \alpha_{i_p}, W_1^*, \ldots, W_m^*, Y_1^*, \ldots, Y_k^*)$ for X, where f is different from the main function symbol of each of $S_1, \ldots, S_n$. This makes inequality axioms out of the delays in which X is delaying variable, whatever substitutions are made for the remaining variables. The substitution is legal by the conditions on delaying variables. ∎

Now for the relation between delays and the constraints described in Section 5. The output of the proof procedure, if it succeeds in proving a free-variable sequent given as input, consists of (i) a set of bindings of free variables, (ii) a set of delayed sequents. Just how this output is produced need not concern us at the moment. We need only know that the delayed sequents conform to the definitions above and that the variable bindings have been obtained by unification. Furthermore, because of substitutions that are made in the algorithm, it will hold that no variable occurs both in a delay and as the left side of a binding. From this output we need to construct an answer in the sense of Section 5, i.e. a set of bindings (possibly empty) of the input variables and a set of constraints (possibly empty) on the auxiliary variables contained in the bindings. For clarity we will speak of the bindings output by the proof procedure as 'unification bindings' and the bindings given in answers (which apply only to input variables) as 'answer bindings'.

We need some definitions. The *input related variables* are an inductively defined subset of the variables occurring in the unification bindings and delays: every input variable is an input related variable, and if $X := S$ is a unification binding and X an input related variable, then every variable in S is input related. Note that all input related variables have restriction value 0. An input related binding is a unification binding $X := S$ where X and hence all variables in S are input related. An input related delay is a variable delay $X \neq S$ where X and all variables in S are input related.

In constructing the answer, we disregard every unification binding and delay that is not input related. The justification for this is the following:

ANSWER LEMMA 1
There is a substitution of terms for the variables that are not input related by which the non-input related delays become inequality axioms whatever the value of the input related variables.

PROOF. This follows from the delay lemma applied to the set of non-input related delays. For no input variable can be delaying variable in a parameter delay, and if an input variable X is delaying variable in a non-input related delay $X \neq S$, we can take a non-input related variable Y in S and make Y the delaying variable instead of X. ∎

Hence we can safely ignore the non-input related unification bindings and delays when extracting an answer to a question.

Now in order to make an answer out of the input related bindings and delays we introduce compositions of the unification bindings and new auxiliary variables as needed so as to get an answer in the sense of Section 5, and produce a constraint from each delay by substituting place markers for the parameters occurring in S. The fairly obvious details (utilizing the fact that unification bindings have been produced with an occur check) are omitted. An example will show the pattern. Suppose we have

input variables X_1, X_2, X_3, X_4 and an output from the proof procedure consisting of the unification bindings $X_1^0 := f(X_2^0, Y^0), Y^0 := g(W^0), X_2^0 := X_3^0$, and the delays $\alpha_3 \neq h(Z_2^5), Z_1^2 \neq g(X_3^0), W^0 \neq f(X_3^0), W^0 \neq f(X_3^0, \alpha_4, \alpha_1)$ (delaying variable W), $W^0 \neq X_3^0$, (delaying variable W), $X_4 \neq Z_5^3$ (delaying variable X_4). Disregarding all that is not input related and dropping the restriction indices (all implicitly 0), we get $X_1 = f(X_2, Y), Y = g(W), X_2 = X_3, W \neq f(X_3, \alpha_4, \alpha_1), W \neq X_3$. Introducing new auxiliary variables and cleaning up, we get $X_1 = f(A, g(W)), X_2 = A, X_3 = A, W \neq f(A, \%1, \%2), W \neq A$ or a variant of this.

The central point is to verify that the delayed sequents will be provable for every instance (in the sense of Section 5) of the answer thus produced. This follows from

ANSWER LEMMA 2
If $t_1, \ldots, t_n$ is an instance of an answer constructed from unification bindings and delays, substitution of $t_1, \ldots, t_n$ for the input variables in an input related delayed sequent yields a sequent provable using the rules and axioms for identity.

PROOF. Suppose the delay is $X \neq S(Y_1, \ldots, Y_k, \alpha_{j_1}, \ldots, \alpha_{j_m})$. A term t which is not of the form $S(s_1, \ldots, s_k, \%1, \ldots, \%m)$, where $s_1, \ldots, s_k$ are the terms substituted for $Y_1, \ldots, Y_k$, will make this delay provable using inequality axioms and the injectivity rule. Again the details are omitted. ∎

8 Rotation and the ordering of premisses

Before giving a full specification of the basic algorithm in Section 9, we will spell out here the rotation scheme (in the sense explained in Section 3) used, and our reasons for using that particular scheme.

First a simpler matter: the order in which to prove the premisses in applications of two-premiss rules. There are four two-premiss rules: $\rightarrow \&, \vee \rightarrow, \supset\rightarrow, \equiv\rightarrow$. In the cases $\rightarrow \&, \vee \rightarrow$ we prove the left premiss first, for no particular reason. In applications of $\supset\rightarrow$ and $\equiv\rightarrow$, however, there is a good reason for proving the right premiss first, viz. that this makes it possible to exploit the semi-invertibility of these rules. If the attempt to prove the right premiss fails we fail the attempted proof of the conclusion. (A precise formulation is given in Section 9.) As it turns out, there is also a good reason for proving the left premiss first, viz. that this accords better with some of the optimizations to be explained below, and this is the principle that has prevailed in the implementations. More on this in later sections. In the basic algorithm described in Section 10, however, the right premiss is proved first in applications of semi-invertible rules.

Our treatment of rotation is the most notably 'mechanical' aspect of our algorithm. To try to prove a sequent in an intelligent way consists to a large extent in scanning the premisses and looking for a premiss which seems suitable to use in order to arrive at the conclusion. Some efficient partial theorem provers like Prolog implement a very specific method of locating a premiss to use in proving a given formula. In the present system, however, we have not implemented any principle of this kind whatsoever. We do not take the conclusion into account at all when choosing a premiss to use.

By a *reduction* we mean an application of an invertible rule other than $\forall \rightarrow$. Because of the limits on contraction, $\forall \rightarrow$ will be treated as a non-invertible rule. The algorithm begins work on a sequent by performing all possible reductions. All left

reductions — i.e. applications of $\& \rightarrow, \vee \rightarrow, \exists \rightarrow$ — are performed first, then all right reductions — applications of $\rightarrow \&, \rightarrow\supset, \rightarrow\equiv, \rightarrow \forall$. An application of $\rightarrow\supset$ or $\rightarrow\equiv$ may introduce the possibility of further left reductions, which are then performed before continuing with any remaining right reductions. This order between left and right reductions does not have any known significance; the only observation we can make here is that there is some slight advantage in applying one-premiss reductions before two-premiss reductions, since this reduces the number of invocations of the proof procedure.

When all possible reductions have been performed we have a sequent $\Gamma \rightarrow C$ where C is atomic or a disjunction or an existential formula, and every formula in Γ is atomic or an implication, an equivalence, or a universal formula. We must have a scheme for using each of these formulae in turn in trying to prove the sequent.

The first principle (of horizontal rotation) we apply is to use the consequent, if it is a disjunction or existential formula, before the antecedent formulae. This principle is dictated by later optimizations (consequent locking). In the basic algorithm it has no other justification than the observation that it seems to work at least as well as other choices.

As regards the antecedent formula, we have further implemented a principle of using the universal formulae before the implications and equivalences. The motivation for this is not theoretical, but consists in the observation that this in practice appears to give very much better results than using implications before universal formulae, and perhaps even better results than letting uncontrolled factors decide the matter. We have not found any theoretical or practical reason for separating implications and equivalences, however.

These are the only horizontal principles. No distinction is made between different universal formulae: they are used in the order in which they happen to have been stored in the universal formula list, and similarly for implications and equivalences.

In the matter of vertical rotation, implications and universal formulae are treated differently. Implications and equivalences are not rotated vertically at all: the implication list is carried over unchanged to the left premiss in an application of $\supset\rightarrow$ or $\equiv\rightarrow$, and in the right premiss we just remove the formula used. In the basic algorithm this entails that in any branch of an attempted proof the formula at the head of the list will be the one to be used as long as any contractions of that formula remain. This is not always a good thing, but the technique of implication locking described in Section 14 makes the vertical non-rotation of implications a reasonable principle.

Universal formulae are rotated vertically as follows:

1. Each sequent has an *active list* and a *dormant list* of universal formulae.
2. When a new universal formula appears (in the examination of the working list as described in Section 9), it is put at the head of the active list.
3. The formulae in the active list are used in order. When a formula from the active list is used it is transferred (if the contraction parameter is not zero) to the dormant list in the premiss sequent.
4. When every formula in the active list has been used, the formulae in the dormant list are transferred to the head of the active list. They will appear in the active list in the order in which they have been used in the branch.

The precise form of this rotation has no very profound motivation. A vertical rotation

of universal formulae is called for, however, since there is nothing corresponding to implication locking in the case of universal formulae. Without any vertical rotation, each universal formula will be used repeatedly in a branch of an attempted proof until no contractions remain, with — as experience has shown — awful results.

The general tendency of the rotation principles is to assume that every premiss in a sequent has its role to play in a derivation of the consequent. When there are many or complicated irrelevant premisses this assumption does not promote efficiency. The technique of use checking (Section 15) counteracts the effect of irrelevant premisses to some extent.

9 Specification of the basic algorithm

The following description concerns the general internal operation of the algorithm. The description of how the algorithm works at top level, begun in Section 5, will be completed in Section 10.

The input to the algorithm consists of a structured and annotated free-variable sequent together with a list of delayed free-variable sequents. The output is either failure or a list of (unification) bindings together with a list of delayed sequents. The algorithm yields multiple output by backtracking: there are a number of choice points, to be pointed out below, where alternatives are systematically tried. When there are no further alternatives, failure is returned.

Just how a free-variable sequent is structured and annotated varies to some extent between different versions and modifications of the algorithm. In all versions, however, sequents have at least the basic structure now to be described.

The sequent is divided into antecedent and consequent. The consequent consists of a single formula. The formulae in the antecedent are distributed over (1) the active and dormant lists of universal formulae, as described in Section 8, (2) a list of implications and equivalences, (3) a list of atomic formulae, (4) the Uf list introduced in Section 6, (5) a list of as yet unanalysed formulae, called the working list. The sequent is also annotated with a parameter index, as explained in Section 6.

Furthermore, the structured sequent includes the contraction annotations for universal formulae, implications, and equivalences. In order to achieve flexibility in the treatment of contraction, we have two ways of determining the number of available contractions of a formula. The first is through an explicit annotation of subformulae of the input formula. The second is by annotating each formula when it is encountered in the working list with an adjustable default contraction value. There is a separate default contraction parameter for universal formulae, implications, and equivalences.

A point concerning substitutions in formulae. Because we always apply the rules from the conclusion to the premisses, we will never make substitutions of bound variables for parameters. We will substitute free variables for bound variables and for parameters and free-variable terms for free variables and for parameters. Thus there is no possibility of variables being captured by quantifiers in substitutions.

Unification occurs, as noted in Section 3, when we are trying to make a logical axiom or equality axiom out of a sequent (whereas the inequality axioms are utilized in the delay mechanism and at a couple of other points as described below). In the case of logical axioms, we have a choice between seeking axioms $A, \Gamma \rightarrow A$ for general formulae A or only for atomic formulae. In the description below, the second

alternative has been chosen, but it is a simple matter to move the check for logical axioms to the beginning of the analysis of the working list. The main reason for restricting axiom checking to atomic formulae is that the unification algorithm can entail very cumbersome computations if there are many equalities in Uf, so we prefer to minimize the applications of unification. A further reduction in the checking for axioms is possible: 8 in the description below is not logically necessary, but can be eliminated.

9.1 The algorithm

For our working sequent, we will use the notation

$$\text{Work,Atoms,Uf,Univs,Imps} \rightarrow C$$

where Work is the working list, Atoms and Uf are as stated above, Univs stands for the pair U_active, U_dormant, and Imps is the list of implications and equivalences. In the following, Γ stands for Atoms,Uf,Univs,Imps. We will use Prolog list notation and write $[A \mid \text{Tail}]$ for a list with first element A, with Tail being the remainder of the list. Similarly $[A, B \mid \text{Tail}]$ is a list with A, B as first two elements.

To make the description of the algorithm more readable, we will use some special terminology for recurring operations.

First, to *continue with* a sequent is to return as output the results of applying the algorithm to the input delays together with that sequent.

The *two-premiss procedure* is activated in the treatment of two-premiss rules. Given the premisses Σ_1 and Σ_2 of a free-variable sequent Σ, we obtain an output by

1. getting an output from the algorithm applied to Σ_1 together with the input delays, and
2. feeding the output delays of step 1 to the algorithm as input delays together with the sequent obtained by performing the substitutions yielded by the output bindings of step 1 in the sequent Σ_2.

As output we deliver the output bindings from step 1 together with the output bindings from step 2 and the delayed sequents output in step 2. We get alternative outputs from the two-premiss procedure by getting alternative outputs from step 2, and when these are exhausted (and thus failure returned in step 2), beginning again with an alternative output from step 1.

The *unify-axiom procedure* is invoked when we seek to produce a logical axiom or equality axiom by unification. Thus, to make an axiom out of $\Gamma \rightarrow S = T$ we seek to unify S and T, giving to the unification algorithm as input the free-variable terms S and T together with the Uf list of the sequent. Similarly in seeking to make a logical axiom out of $A, \Gamma \rightarrow B$. Given an output of bindings from the unification algorithm we apply a generalization of the two-premiss procedure to the input delayed sequents $\Sigma_1 \ldots, \Sigma_k$. Explicitly: for $i = 1, \ldots, k$ we determine an input and output as follows:

- for $i = 1$ (step 1), we input the empty list of delays, and the sequent Σ_1' obtained by making the substitutions yielded by the unification in the sequent Σ_1, and output the unification bindings and either Σ_1' if the delaying variable of Σ_1 has not been bound, or the result of applying the algorithm to Σ_1'.

- for $i > 1$, we input the output delays of step $i-1$ and the sequent Σ_i' obtained by making the substitutions corresponding to the bindings output in step i in the sequent sequent Σ_i, and output the union of the lists of bindings output in the previous steps together with either Σ_i', if the delaying variable of Σ_i' has not been bound, or the result of applying the algorithm to Σ_i'.

Here we get alternative outputs in analogy to the two-premiss case, but in addition we get alternatives by getting alternative output from the unification that started the whole thing.

The *substitution procedure* operates on a sequent and a pair of terms, one of which is a parameter. We substitute the other term for the parameter throughout the sequent. Next we check the resulting Uf list to see if it has become possible to delay the sequent as a result of the substitution. If so, we output the empty list of bindings together with the input delays and the present sequent as new delay. If the sequent has not become delayable after the substitution, we continue with that sequent.

Now for the description of the procedure. If the working list is [A |Work] and thus non-empty, we begin by determining the form of A and taking various actions. The numbered cases below represent actions and alternative actions, as indicated.

1. A is $\bot$. In this case we return the empty list of bindings together with the input list of delays.
2. A is $A_1 \& A_2$. We continue with the sequent

$$[A_1, A_2 \mid \text{Work}], \Gamma \rightarrow C$$

3. A is $A_1 \vee A_2$. We apply the two premiss procedure to $[A_1 \mid \text{Work}], \Gamma \rightarrow C$ and $[A_2 \mid \text{Work}], \Gamma \rightarrow C$.
4. A is $A_1 \supset A_2$ or $A_1 \equiv A_2$, possibly with a contraction annotation. If A is not already annotated, we annotate it with the default implication or equivalence annotation. We continue with the sequent

$$\text{Work, Atoms, Uf, Univs, } [A \mid \text{Imps}] \rightarrow C$$

5. A is $\exists x B(x)$. We continue with the sequent

$$[B(\alpha_i) \mid \text{Work}], \text{ Atoms, Uf, Univs, Imps} \rightarrow C$$

   where i is the current parameter index, and the parameter index of the indicated sequent is set to $i+1$.
6. a is $\forall x B(x)$, possibly with a contraction annotation. If A is not already annotated, we annotate it with the default contraction value for universal formulae. We continue with the sequent

$$\text{Work,Atoms,Uf,}[A \mid \text{U\_active}],\text{U\_dormant,Imps} \rightarrow C$$

7. A is an equality $S = T$. There are a number of mutually exclusive cases.
   (a) A is $X = X$ for some variable X. In this case we continue with Work, $\Gamma \rightarrow C$.
   (b) A is $X = S$ where S properly contains the variable X. In this case we return the empty list of bindings together with the input list of delays.

(c) A is $X^n = \alpha_k$ where $k \geq n$. We apply the substitution procedure to Work, $\Gamma \rightarrow C$ and the substitution of X^n for α_k.

(d) A is $X = S$ where none of $a-c$ applies. In this case we add our working sequent to the input list of delays as a new variable delay with X as delaying variable. We return the resulting list of delays together with the empty list of bindings.

(e) A is $S = X$ where S is not a variable. We continue with $[X = S \mid \text{Work}], \Gamma \rightarrow C$.

(f) A is $\alpha_n{=}\alpha_n$ In this case we continue with Work $\Gamma \rightarrow C$.

(g) A is $\alpha_i{=}\alpha_j$ where $i \neq j$. We continue with the result of substituting α_i for α_i (if $i < j$) or α_j for α_j (if $j < i$) everywhere in Work,$\Gamma \rightarrow C$.

(h) A is $\alpha_n = S$ where S properly contains α_n. In this case we return the empty list of bindings together with the input list of delays.

(i) A is $\alpha_n = S$ where S contains an n-admissible variable. In this case we add our working sequent to the input list of delays as a new parameter delay with the n-admissible variable found as delaying variable. We return the resulting list of delays together with the empty list of bindings.

(j) A is $\alpha_n = S$ where none of $f - i$ applies. We apply the substitution procedure to Work, $\Gamma \rightarrow C$ with the substitution of S for α_n, but with the following added twist: if S contains a parameter α_k where $k > n$, we add $\alpha_n = S$ to the Uf list and do not substitute for this particular occurrence of α_n.

(k) A is $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)$, with different function symbols f, g $(m, n \geq 0)$. In this case we return the empty list of bindings together with the input list of delays.

(l) A is $f(s_1, \ldots, s_m) = f(t_1, \ldots, t_m)$. In this case we continue with the sequent $[s_1 = t_1, \ldots, s_m = t_m \mid \text{Work}], \Gamma \rightarrow C$.

8. A is an atomic formula other than an equality. Here we apply the unify-axiom procedure to A and the consequent C. An alternative is given in 9.

9. As an alternative to 8 we continue with the sequent

$$\text{Work},[A \mid \text{Atoms}],\text{Uf},\text{Univs},\text{Imps} \rightarrow C$$

If the working list is empty, we turn our attention to the consequent C in the sequent $\Gamma \rightarrow C$. In the following cases 10–20 it is assumed that the working list is empty.

10. If C is $C_1 \& C_2$ apply the two-premiss procedure to $\Gamma \rightarrow C_1, \Gamma \rightarrow C_2$.

11. If C is $C_1 \supset C_2$ continue with $[C_1], \Gamma \rightarrow C_2$, where $[C_1]$ is the working list.

12. If C is $C_1 \equiv C_2$, apply the two-premiss procedure to $[C_1], \Gamma \rightarrow C_2$ and $[C_2], \Gamma \rightarrow C_1$.

13. If C is $\forall x A(x)$, we continue with the sequent

$$\Gamma \rightarrow A(\alpha_i)$$

where i is the current parameter index, and the parameter index of the indicated sequent is set to $i + 1$.

14. If C is $C_1 \vee C_2$ we continue with $\Gamma \rightarrow C_1$. As an alternative, we continue with $\Gamma \rightarrow C_2$. A further alternative is given in 18.

15. If C is $\exists x A(x)$ we introduce a new variable X^i where i is the current parameter index, and continue with $\Gamma \rightarrow A(X^i)$. An alternative is given in 18.

16. If C is an equality $S = T$, we apply the unify-axiom procedure to S and T. An alternative is given in 18.
17. If C is atomic formula, we apply the unify-axiom procedure to C and a formula in the Atoms list. Each choice of formula in the Atoms list gives an alternative output. A further alternative is given in 18.
18. We use each universal formula $\forall x A(x)$ in the active list to obtain outputs, by introducing a new variable X^i where i is the current parameter index and applying the procedure to the input delays and the sequent

$$[A(X^i)], \text{Atoms}, \text{Uf}, \text{Univs}', \text{Imps} \rightarrow C$$

Here Univs′ is Univs with $\forall x A(x)$ removed if no contractions remain, or otherwise with its contraction count decremented by one and moved to the head of the dormant list. An alternative is given in 19 .

19. We use an implication $A \supset B$ or equivalence $A_1 \equiv A_2$ in the implication list to obtain an output by applying the two-premiss procedure to the corresponding premiss sequents, taking the right premiss first.
Thus in the case of an implication we have right premiss $[B]$,Atoms, Uf, Univs, Imps′ $\rightarrow C$, where Imps′ is Imps with $A \supset B$ removed, and left premiss Atoms, Uf, Univs, Imps′ $\rightarrow A$, where Imps′ is Imps with $A \supset B$ removed if its current contraction count is 1, or otherwise with the contraction count of $A \supset B$ decremented by one. In the case of an equivalence we have the right premiss $[A_1, A_2]$, Atoms, Uf, Univs, Imps′ $\rightarrow C$ and left premiss Atoms, Uf, Univs, Imps′ $\rightarrow A_i$.
Each choice of implication or equivalence yields an alternative output. The choice of A_i to use in the left premiss also gives alternatives in the case of the $\equiv\rightarrow$-rule. The following special rule applies, however: if for any of the implications or equivalences the right premiss cannot be proved at all — i.e. the first time we apply the procedure to that premiss, failure is returned — we do not continue with the remaining implications and equivalences, but go directly to the alternative given in 20. (This is where we use the semi-invertibility of these rules.)
20. This alternative to 19 consists in reinstating the universal formulae in the dormant list as described in Section 8 and continuing with the resulting sequent.

10 Top level procedure

Given a question in the form of a free-variable formula $A(X_1, \ldots X_n)$ optionally annotated with contraction values, we apply the following procedure to produce a series of answers.

Form a starting sequent by setting the consequent equal to $A(X_1^0, \ldots X_n^0)$, the antecedent lists being empty. Feed this sequent with the initial parameter index 0 to the algorithm specified in Section 9, together with the empty list of delays. There are two matters to be considered: what to do with an individual output returned by the algorithm, and how to prompt for further outputs.

First the treatment of individual outputs. If the algorithm yields a non-failure output, this output is transformed into an answer as described in Section 5, and possibly presented as output of the top level procedure. Every answer presented is saved in a database of answers. Whether or not an answer derived from the algorithm

is presented as output depends both on whether it returns any bindings and on how thorough we are in our subsumption checking.

The strictest possible criterion for presenting an answer is that not every instance of that answer (in the sense of Section 5) is an instance of some previously presented answer. For example, if we have previous solutions

1. $X = f(g(Y))$ and
2. $X = f(Y)$ where $Y \neq g(\%1)$,

the strict criterion will eliminate a later answer $X = f(Y)$, since every instance of that answer is an instance of one of the previous answers. However, since it may not be immediately obvious to the user of the system in such cases that the new answer is subsumed by previous answers, one may well prefer to present the answer, and use the subsumption checking only to weed out dreary repetitions or obvious variants of previously given answers. Since we have at present no well thought-out algorithm for doing strict subsumption checking, we leave the matter open.

The second question to be considered is how to get further answers. We must of course prompt the algorithm to produce every output of which it is capable given the many choice points indicated, where alternative outputs may be sought. At the top level there are, however, further alternatives to be considered. In the algorithm as described in Section 9 there are no alternatives to delaying a sequent. There are alternatives to all operations in which variables are bound, for even if there are no input variables we must in general find every binding of the internally generated free variables that yields a provable instance of a free-variable formula. (Consider for example an attempted proof of a sequent of the form $\Gamma \rightarrow \exists x(A(x)\&B(x))$, where every binding of X for which $\Gamma \rightarrow A(X)$ is provable may have to be found). Delays, however, bind no variables, and it would in general be a waste of time to investigate alternatives to making those delays. At top level, on the other hand, we must consider alternatives to delays in order to obtain a complete algorithm in the sense of Section 5.

For example, the question $\forall y(X = g(y) \supset \forall z(\neg X = f(z)))$ leads to the sequent

$$X^0 = g(\alpha_0) \rightarrow \forall z(\neg X^0 = f(z))$$

This sequent will be delayed without further ado, and we extract the answer

$$X = A \text{ where } A \neq g(\%1)$$

Clearly this answer is not sufficient, however, since the formula is in fact valid for every substitution of a closed term for X.

We therefore adopt the following procedure. When an answer has been extracted from an output we check for alternatives to each of the input related delays in the output. These alternatives are generated by taking an input related delay $X = S, \Gamma \rightarrow C$ and attempting to prove the sequent $\Gamma' \rightarrow C'$ obtained by substituting S for X everywhere in $\Gamma \rightarrow C$. (Here we utilize the fact that general replacement holds as a derived rule in GF.) Thus in the above example, we try to prove $g(\alpha_0) = g(\alpha_0) \rightarrow \forall z(\neg g(\alpha_0) = f(z))$ and succeed with an output of no bindings and no delayed sequents; hence we report an answer with X unbound and unconstrained.

Thus the top level procedure for extracting answers consists in applying the algorithm with the invocation of additional alternatives at certain points. Because of the limited contraction, only a finite number of answers can be generated for a given default contraction value and a given annotation of the input formula. In order to generate further answers we increase the default contraction value and start over. Of course, if all universal formulae, implications, and equivalences in the input are annotated with contraction values, there is no point in starting over with new default values.

11 Locking the consequent

The large number of choice points is a striking aspect of the algorithm described. Clearly the alternatives multiply at a computationally very unpleasant rate. In the following sections we will present several *restrictions* of the algorithm. By this we mean a modification of the algorithm which consists in cutting away some alternatives. A particularly unproblematic form of restriction is that which we will call a *pruning*: an elimination of alternatives that cannot, in the circumstances in which the pruning is applied, lead to any new successful bindings. The only way in which a pruning can slow down the algorithm is by consuming more time in checking that the necessary conditions are satisfied than is gained by avoiding unnecessary computations. Most useful restrictions turn out not to be prunings. For example, using invertible rules before non-invertible rules is an example of a restriction that is not a pruning. Of course for the algorithm to remain complete, a restriction must not eliminate every way in which a particular successful binding can be found. For any restriction that is not a pruning there will be cases where proofs that would have been easy to find without the restriction are eliminated in favour of proofs that are hard to find. Nevertheless, in practice it appears that the high degree of non-determinism is so detrimental to the performance of the algorithm that the overall effect even of quite drastic restrictions, by no means obviously completeness preserving, is almost wholly beneficial. Some of the restrictions to be described entail that formulae will sometimes require higher contraction values to be provable than are necessary when the basic algorithm is used.

The restrictions presented in the present section are, however, prunings: $\vee$-*locking* and $\exists$-*locking*. They work as follows. Before seeking alternatives to using a disjunction or existential formula in the consequent we annotate the sequent with a flag indicating that the consequent of the sequent is *locked*. That the consequent is locked means that we are not allowed to apply a consequent rule to the sequent. The consequent is unlocked — i.e. the annotation is removed — at any application of the $\vee \rightarrow$-rule, and in the left branch of an application of $\supset\rightarrow$ or $\equiv\rightarrow$. In addition, if the locked consequent is an existential formula, it is unlocked an any application of $\exists \rightarrow$. It should be noted that this optimization exploits the sequential nature of the algorithm and is not obviously applicable when different branches of an attempted proof are explored in parallel.

Locking the consequent is an inexpensive pruning and will give good results when added to the basic algorithm. However, in practice it is seldom of importance because the effects of locking are usually subsumed by those of compaction, which is an important and indeed indispensable form of restriction in our algorithm.

12 Compaction

In seeking a proof of a sequent $\Gamma \rightarrow (A \vee B) \vee C$ we will, in the basic algorithm, first try to prove $\Gamma \rightarrow A \vee B$. This in turn leads us to the task of proving $\Gamma \rightarrow A$. We thus have two steps in the attempted proof:

$$\dfrac{\dfrac{\Gamma \rightarrow A}{\Gamma \rightarrow A \vee B}}{\Gamma \rightarrow (A \vee B) \vee C}$$

To apply compaction in this case is to squeeze these two steps into one:

$$\dfrac{\Gamma \rightarrow A}{\Gamma \rightarrow (A \vee B) \vee C}$$

The point of this is not to reduce the number of invocations of the proof procedure from three to two, but to do away with a large number of alternatives: all of the alternative proofs of $\Gamma \rightarrow A \vee B$ using formulae in Γ that will be attempted in the basic algorithm are eliminated. We retain the alternative of proving $\Gamma \rightarrow B$ and also the alternative attempted proofs of $\Gamma \rightarrow (A \vee B) \vee C$ using either C as consequent or one of the formulae in Γ, and for the algorithm to remain complete these must suffice to obtain every successful binding otherwise yielded by the eliminated deductions.

The above type of compaction will be called $\vee$-$\vee$-compaction. Various forms of compaction are possible. Those that seem to notably increase the efficiency of the basic algorithm are, in addition to $\vee$-$\vee$-compaction, $\vee$-$\exists$-compaction, $\exists$-$\vee$-compaction, $\exists$-$\exists$-compaction, $\forall$-$\supset$-compaction, $\forall$-$\equiv$-compaction, and $\forall$-$\forall$-compaction, all of them defined below. The use of these forms of compaction is most simply described as a modification of some of the logical inference rules.

We define the $\vee\exists$-*expansions* of a formula A inductively as follows: A is a $\vee\exists$-expansion of A; if $B(t)$ is a $\vee\exists$-expansion of A then $\exists x B(x)$ is a $\vee\exists$-expansion of a; if B is a $\vee\exists$-expansion of A and C is any formula, $B \vee C$ and $C \vee B$ are $\vee\exists$-expansions of A.

To implement $\vee$-$\vee$-compaction, $\vee$-$\exists$-compaction, $\exists$-$\vee$-compaction, and $\exists$-$\exists$-compaction we replace the $\rightarrow \vee$-rule and $\rightarrow \exists$-rule by the following rule:

$$\dfrac{\Gamma \rightarrow A}{\Gamma \rightarrow B}$$

where A is not a disjunction or an existential formula, and B is a $\vee\exists$-expansion of A. In applying this rule (from the conclusion to the premiss) in the algorithm, we take as alternatives every $\Gamma \rightarrow A$ where B is a $\vee\exists$-expansion of A, before going on to use the formulae in Γ.

To implement $\forall$-$\supset$-compaction, $\forall$-$\equiv$-compaction, and $\forall$-$\forall$-compaction we introduce three new rules of inference to replace the $\forall \rightarrow$-rule:

$$\dfrac{D, A(t_1, \ldots, t_n) \supset B(t_1, \ldots, t_n), \Gamma \rightarrow A \quad D, B(t_1, \ldots, t_n), \Gamma \rightarrow C}{\forall x_1 \ldots \forall x_n (A(x_1, \ldots, x_n) \supset B(x_1, \ldots, x_n)), \Gamma \rightarrow C}$$

where D is $\forall x_1 \ldots \forall x_n (A(x_1 \ldots x_n) \supset B(x_1 \ldots x_n))$.

$$\dfrac{D, A_1(t_1, \ldots, t_n) \equiv A_2(t_1, \ldots, t_n), \Gamma \rightarrow A_i \quad D, A_1(t_1, \ldots, t_n), A_2(t_1, \ldots, t_n), \Gamma \rightarrow C}{\forall x_1 \ldots \forall x_n (A_1(x_1, \ldots, x_n) \equiv A_2(x_1, \ldots, x_n)), \Gamma \rightarrow C}$$

where D is $\forall x_1 \ldots \forall x_n (A_1(x_1, \ldots, x_n) \equiv A_2(x_1, \ldots, x_n))$.

$$\frac{\forall x_1 \ldots \forall x_n A(x_1, \ldots, x_n), A(t_1, \ldots, t_n), \Gamma \rightarrow C}{\forall x_1 \ldots \forall x_n (x_1, \ldots, x_n), \Gamma \rightarrow C}$$

In the last of these rules $A(x_1, \ldots, x_n)$ must not be a universal formula, an implication, or an equivalence.

The algorithm is adapted to these rules in the fairly obvious fashion. Note that the modified $\forall \rightarrow$-rule is semi-invertible in the first two cases.

13 Sifting bindings

The two-premiss procedure described in Section 9 has the following defect: in seeking alternative proofs of the first premiss, we don't check whether the bindings yielded by those alternatives are in fact new. Because of this we may spend a lot of time trying to prove the second premiss over and over, with the same set of bindings. Hence we introduce a sifting of bindings now to be described, and to be called *two-premiss sifting.*

Let Σ_1 and Σ_2 be the premisses of Σ in an application of the two-premiss procedure. The following sifting principles apply in two-premiss sifting:

1. Every output — bindings and delays — returned by alternative proofs of Σ_1 is saved and then labelled 'success' or 'failure' depending on whether or not the corresponding attempted proof of Σ_2 succeeds. On backtracking to the attempted proof of Σ_1 and obtaining another solution of Σ_1, we compare that solution with the stored previous solutions, and reject it if there is a previous solution such that
   (a) the delays of that solution form a subset of the delays of the present solution, and
   (b) the previous solution was a success and the current solution is an instance of that solution, or the previous solution was a failure and the current solution is an instance of that solution with respect to the free variables in Σ_2.
2. If the *first* attempt to prove Σ_2 using bindings and delays returned by a proof of Σ_1 returns failure, check whether both of the following conditions are satisfied:
   (a) there is no binding of any variable in Σ_2 in the bindings returned
   (b) no delaying variable in any of the delays returned by the proof of Σ_2 occurs in Σ_1.

   If both conditions are satisfied, we know that alternative proofs of Σ_1 cannot help us prove Σ_2, and so we do not seek any such alternative proofs.

Note that even if conditions (a) and (b) in (2) are satisfied, we must still backtrack to Σ_1 if an attempted proof of Σ_2 fails in the course of backtracking, rather than on the first try. It is, however, inexpensive to incorporate a further check of proofs of Σ_1: if Σ_1 is proved without binding any variable whatever (which is not infrequently the case) and without returning any delays, we know that we need not seek alternative proofs of Σ_1 under any circumstances. This further check has been incorporated in at least one implementation.

Two-premiss sifting also applies to proving delayed sequents in the unify-axiom-procedure. We omit the details.

Clearly two-premiss sifting is not the only sifting that can be introduced into the algorithm. We have in fact experimented with what may be called *choice point sifting*: at various choice points in the algorithm, solutions are stored and on backtracking it is ensured that only essentially new solutions are accepted. However, experience indicates that choice point sifting quickly becomes very expensive and yields only meagre results, whereas the form of two-premiss sifting specified above is a highly successful optimization of the algorithm.

14 Implication locking

Implication locking is a technique for drastically restricting the use of implications and equivalences in the antecedent. First some definitions.

An application of $\rightarrow\supset$ will be called a *transfer*; the antecedent A of the principal formula $A \supset B$ in the application will be called the *transferred* formula. We extend the structure of sequents by adding a *transfer list* to the other lists. In each transfer, the transfer list of the premiss is extended to contain the transferred formula.

To explain implication locking, we start by considering the simpler case of propositional logic. Here there are two principles:

1. If, in an application of $\rightarrow\supset$ to a sequent $\Gamma \rightarrow A \supset B$ the formula A already occurs in the transfer list of the sequent, we continue with the sequent $\Gamma \rightarrow B$; and similarly in applications of $\rightarrow\equiv$.
2. If, after an application of $\supset\rightarrow$ or $\equiv\rightarrow$ to a sequent $\Gamma \rightarrow C$, we come in the left premiss branch, via applications of consequent rules, to $\Gamma \rightarrow C'$ (no new formula having been transferred) and thus should, according to the basic algorithm, use an implication or equivalence in Γ, we throw away instead everything done after the initial application of $\supset\rightarrow$ or $\equiv\rightarrow$, and start on the next alternative to that application.

As a simple example to see what this entails, consider the sequent

$$p_1 \supset p_2, p_2 \supset p_3, \ldots, p_{n-1} \supset p_n, p_1 \rightarrow p_n$$

This sequent can be proved using the premiss implications in any order and is handled very quickly by the algorithm without implication locking. However since there will be no transfer at all in a proof of the sequent, implication locking constrains the implications to be used in one particular order, namely that exhibited above. Thus in this case we have a drastic restriction of the algorithm whereby only one of out of $n!$ possible proofs remains to be found.

This extreme example illustrates both the power and the possible drawbacks of restrictions. In fact the above sequent will take some considerable time to prove for large n using the algorithm described, so it would appear that implication locking is not always beneficial. However, the trouble with this example is not due to the implication locking as such, but to the implication locking in combination with the principle that the right premiss will be proved first in applications of $\supset\rightarrow$ or $\equiv\rightarrow$ to. Accordingly, implication locking must be coupled with the stipulation that the premisses in applications of these rules will be proved in the other order, the left premiss being proved first. This completely eliminates the drawbacks of implication

locking. (The fact that a shorter derivation may be rejected in favour of a longer one when implication locking is used is not significant in practice since non-determinism is our main problem.) In the propositional case we can still make use of the semi-invertibility of $\supset\!\!\rightarrow$ and $\equiv\!\!\rightarrow$ whenever the attempted proof of the right premiss fails.

In the predicate logic case we also use a transfer list, and principle 1 remains as above. The second principle is somewhat different:

- Suppose we apply $\supset\!\!\rightarrow$ or $\equiv\!\!\rightarrow$ to a sequent $\Gamma \rightarrow C$ and then come to $\Gamma \rightarrow C'$, without having transferred any new formula (i.e. one not in the transfer list). We then insert a *barrier* at the head of the implication list in Γ before continuing with $\Gamma \rightarrow C'$. A barrier in the implication list has the effect of preventing the use of any implication or equivalence after the barrier. Whenever a new formula is transferred, the *first* barrier is removed from the implication list.

The use of implication locking in the predicate logic algorithm is also coupled with a left to right order in proving the premisses of $\supset\!\!\rightarrow$ or $\equiv\!\!\rightarrow$. These remarks also apply to allimp and alliff compaction. In the predicate logic case we can still incorporate a certain exploitation of semi-invertibility in two-premiss sifting, as follows: if the left premiss has been proved without binding any variable in the right premiss, and the attempted proof of the right premiss fails, we reject the parent sequent.

15 Use checking

In order to weed out some of the irrelevant computations that are inevitable when proofs are sought — particularly, of course, if irrelevant premisses appear at an early stage — the following checks can be made in applications of two-premiss rules.

To make *essential use* of an occurrence of a formula in proving a sequent is to use some subformula of that occurrence as the formula A in a logical axiom $A, \Gamma \rightarrow A$. Thus this notion presupposes that we work with analysed proofs, i.e. proofs in which each occurrence of a formula is kept track of across the proof.

1. $\supset\!\!\rightarrow$-use checking: If, in proving a sequent $A \supset B, \Gamma \rightarrow C$ we apply $\supset\!\!\rightarrow$ to the formula $A \supset B$ and prove the right premiss $B, \Gamma \rightarrow C$ without making essential use of the indicated occurrence of the formula B, we return the resulting bindings and delays as output of the attempted proof, without trying to prove the left premiss.
2. $\equiv\!\!\rightarrow$-use checking: If, in proving a sequent $A \equiv B, \Gamma \rightarrow C$ we apply $\equiv\!\!\rightarrow$ to the formula $A \equiv B$ and prove the right premiss $A, B, \Gamma \rightarrow C$ without making essential use of either of the indicated occurrences of the formulae A, B, we return the resulting bindings and delays as output of the attempted proof, without trying to prove the left premiss.
3. $\rightarrow$ &-inconsistency checking: If, in proving a sequent $\Gamma \rightarrow A\&B$ we prove $\Gamma \rightarrow A$ without making essential use of the indicated occurrence of A, we return the resulting output as output of the attempted proof of $\Gamma \rightarrow A\&B$ without proving the right premiss. (Note that in this case Γ must be inconsistent.)
4. $\vee \rightarrow$-use checking: If, in proving a sequent $A \vee B, \Gamma \rightarrow C$ we prove $A, \Gamma \rightarrow C$ without making essential use of the indicated occurrence of A, we return the

resulting output as output of the attempted proof of $A \vee B, \Gamma \rightarrow C$ without proving the right premiss.

However, (1) and (2) presuppose that the right premiss is proved first in applications of $\supset\rightarrow$ and $\equiv\rightarrow$, and hence are not available in the optimized algorithm. Instead we apply the inconsistency check applied to $\rightarrow$ & also in the case of $\supset\rightarrow$ and $\equiv\rightarrow$:

5. $\supset\rightarrow$-inconsistency checking: If, in proving a sequent $A \supset B, \Gamma \rightarrow C$ we apply $\supset\rightarrow$ to the formula $A \supset B$ and prove the left premiss $A \supset B, \Gamma \rightarrow A$ without making essential use of the indicated occurrence of the formula A, we return the resulting bindings and delays as output of the attempted proof, without trying to prove the right premiss.
6. $\equiv\rightarrow$-inconsistency checking: If, in proving a sequent $A \equiv B, \Gamma \rightarrow C$ we apply $\equiv\rightarrow$ to the formula $A \equiv B$ and prove the left premiss $A \equiv B, \Gamma \rightarrow D$ (where D is A or B) without making essential use of the indicated occurrence of the formula D, we return the resulting bindings and delays as output of the attempted proof, without trying to prove the right premiss.

(3)–(6) have been implemented for propositional logic, to good effect.

16 Collecting implications

Intuitionistic logic does not offer many possibilities of rewriting sequents into equivalent and more easily handled forms. The two obvious ways of combining implications in the antecedent have, however, turned out to be of some use.

The first of these operations consists in checking an implication $A \supset B$ whenever it appears in the course of the analysis of the working list to see if an implication $A \supset C$ already exists in the implication list. If so we replace $A \supset C$ in the implication list by $A \supset B\&C$, taking the contraction value of this implication to be the maximum of the separate values of $A \supset C$ and $B \supset C$.

The use of conjunction is just a matter of convenience. Essentially, we are introducing a rule

$$\frac{A \supset B_1, \ldots, A \supset B_n, \Gamma \rightarrow A \quad B_1, \ldots, B_n, \Gamma \rightarrow C}{A \supset B_1, \ldots, A \supset B_n, \Gamma \rightarrow C}$$

in addition to the standard $\supset\rightarrow$-rule.

The second operation works similarly, but looks for implications $A \supset C$ and $B \supset C$ to combine into $A \vee B \supset C$. In other words, we introduce a rule

$$\frac{A_1 \supset B, \ldots, A_n \supset B, \Gamma \rightarrow A_i \quad B, \Gamma \rightarrow C}{A_1 \supset B, \ldots, A_n \supset B, \Gamma \rightarrow C}$$

Again it is convenient to use rewriting in implementing this rule. In this case, however, we use a special connective $\sigma\vee$ (splitting disjunction) for which we have a rule to the effect that $\Gamma \rightarrow A\sigma\vee B$ can *only* be proved by proving one of $\Gamma \rightarrow A$ or $\Gamma \rightarrow B$, and rewrite $A \supset B$ and $C \supset B$ as $A\sigma\vee C \rightarrow B$. The further alternatives that would be tried with an ordinary disjunction are clearly redundant here.

17 A propositional decision procedure and its use in prop checking

In applying the algorithm to propositional formulae, simplifications can be made. The entire delay mechanism falls away, and everything having to do with binding variables. Unification reduces to a simple comparison of atomic formulae for equality. Input to the procedure consists only of a sequent, and the output of success (formerly the empty list of bindings) or failure. In the two-premiss procedure there is no need to consider any alternatives to a proof of Σ_1 if the proof of Σ_2 fails; instead we just return failure as output of the procedure. Alternatives remains to be considered only in proving a sequent $\Gamma \rightarrow C$ where C is atomic or a disjunction and Γ consists of atomic formulae, implications, and equivalences: here we must be prepared to try using each of the non-atomic formulae. Implication locking, which is essential for the feasibility of the algorithm, eliminates the need for contraction deepening, since no implication $A \supset B$ can be used more times than there are subformulae to transfer from A (and similarly for equivalences). Thus we obtain a decision procedure for propositional intuitionistic logic. Of the other optimizations introduced above, $\vee$-$\vee$-compaction, $\rightarrow$ &-inconsistency checking, $\vee$-checking, $\supset\rightarrow$-inconsistency checking and $\equiv\rightarrow$-inconsistency checking, and combining implications carry over immediately to propositional logic. All except combining implications have been found to contribute greatly to the efficiency of the algorithm.

The resulting decision procedure has a different character to that of [6] in not attempting any reduction of a problem in propositional logic to a problem involving (a large number of) simpler formulae. Instead it is based on exploiting the complexity of formulae.

In the implementation of the predicate logic algorithm, propositional input is given to an implementation of this propositional decision procedure, which decides the formula very much faster than the predicate logic algorithm will succeed or fail at a given contraction depth. This routine has also been used to implement an optimization called *prop checking*, which simply consists in checking at various places that the propositional skeleton of a sequent — i.e. the result of deleting all quantifiers and variables — is valid in propositional logic. More precisely, given that an initial sequent is propositionally valid, it needs to be checked that the working premiss is still propositionally valid at applications of $\rightarrow\vee$, and in proving the left premiss of $\supset\rightarrow$ or $\equiv\rightarrow$ (and in the corresponding cases of compaction). This is a crude optimization, particularly as implemented in the present system, but it is of interest as an aid in investigating to what extent the algorithm is slowed down by the fact that it does not look for relevant premisses. As it turns out, prop checking will sometimes drastically reduce the time it takes for the algorithm to deal with problems containing many irrelevant premisses, but usually just adds some overhead in dealing with more structured problems.

References

[1] D. van Dalen, Intuitionistic logic. In *Handbook of Philosophical Logic, Vol. III*, eds D. M. Gabbay and F. Guenthner, D. Reidel, 1986.

[2] G. K. Dardzania, Intuitionistic system without contraction. *Bulletin of the Section of Logic, Polish Academy of Sciences, Institute of Philosophy and Sociology*, **6**, 1977.

[3] M. Fitting, Resolution for intuitionistic logic. Paper presented at ISMIS '87, Charlotte, NC. 1987.

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

[5] S. Kanger, A simplified proof method for elementary logic. In *Computer Programming and Formal Systems*, North-Holland, 1963.

[6] B. B. Volozh *et al.*, The Priz system and propositional calculus. Translated from *Kibernetika*, **6** 1982.

Appendix

Implementations of the algorithm in SICStus Prolog (which is Quintus compatible) and in *C* (the latter for 4.2BSD or 4.3BSD) are available by anonymous ftp from sics.se, IP address 192.16.123.90. The implementations can also be obtained by email: send your request to either dan@sics.se or torkel@sics.se.

The execution times given (in milliseconds) in the following benchmarks pertain to the *C* implementation (running on a Sparc 1 work station), which does not incorporate any treatment of identity; hence the omission of identity logic from the examples. The timing 'awful' or 'hopeless' may be interpreted as any unacceptable value. All examples have been given to the algorithm with the following settings (the other optimizations mentioned above not being active):

1. all forms of compaction active,
2. implication locking, or-locking, and existence-locking active,
3. two-premiss sifting active,
4. initial contraction parameters all have value 1, and all are incremented by 1 on failure.

The examples have not been chosen with a view to possible applications of intuitionistic logic, but only in order to test various aspects of the algorithm. The parameters of the algorithm have been kept constant, but in many cases, as noted below, a result would have been obtained either very much earlier or very much later with different settings. It will also be seen from the comments that even a modicum of parallelism can be expected to contribute greatly to the efficiency of the algorithm, in view of its current extreme sensitivity to the precise formulation of the premisses.

1. Alternations of quantifiers.

1.1 $\forall x\exists y\forall z(p(x) \,\&\, q(y) \,\&\, r(z)) \leftrightarrow \forall z\exists y\forall x(p(x) \,\&\, q(y) \,\&\, r(z))$ [120]

1.2 $\forall x\exists y\forall z\exists w(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w)) \leftrightarrow \exists w\forall z\exists y\forall x(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w))$ [2050]

1.3 $\forall x\exists y\forall z\exists w\forall u(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w) \,\&\, t(u)) \leftrightarrow$
$\forall u\exists w\forall z\exists y\forall x(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w) \,\&\, t(u))$ [awful]

1.4 $\exists z\forall x\exists y(p(x) \,\&\, q(y) \,\&\, r(z)) \leftrightarrow \exists y\forall x\exists z(p(x) \,\&\, q(y) \,\&\, r(z))$ [40]

1.5 $\exists z\forall x\exists y\forall w(p(x) \,\&\, q(y) \,\&\, r(z)) \leftrightarrow$
$\forall w\exists y\forall x\exists z(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w))$ [7050]

1.6 $\exists z\forall x\exists y\forall w\exists u(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w) \,\&\, t(u)) \leftrightarrow$
$\exists u\forall w\exists y\forall x\exists z(p(x) \,\&\, q(y) \,\&\, r(z) \,\&\, s(w) \,\&\, t(u))$ [74290]

1.7 $\exists x1\forall y1\exists x2\forall y2(p(x1, y1) \,\&\, q(x2,y2)) \supset \forall y2\exists x2\forall y1\exists x1(p(x1,y1) \,\&\, q(x2,y2))$ [560]

1.8 $\exists x1\forall y1\exists x2\forall y2\exists x3\forall y3(p(x1,y1) \,\&\, q(x2,y2) \,\&\, r(x3,y3)) \supset$
$\forall y3\exists x3\forall y2\exists x2\forall y1\exists x1(p(x1,y1) \,\&\, q(x2,y2) \,\&\, r(x3,y3))$ [awful]

2. Append

2.1 $\forall x$ append(nil,x,x) &
$\forall x\forall y\forall z\forall w$(append(y,z,w) $\supset$ append(cons(x,y),z,cons(x,w)))
$\supset$
$\exists x$ append(cons(a1,cons(a2,cons(a3,cons(a4,cons(a5,nil))))),nil,x) [7720]

2.2 $\forall x$ append(nil,x,x) &
$\forall x\forall y\forall z\forall w$(append(y,z,w) $\supset$ append(cons(x,y),z,cons(x,w)))
$\supset$
$\exists x$ append(cons(a1,cons(a2,cons(a3,cons(a4,cons(a5,cons(a6,nil)))))),nil,x) [15759]

2.3 ∀x append(nil, x,x) &
∀x∀y∀z∀w(append(y,z,w) ⊃ append(cons(x,y),z,cons(x,w)))
⊃
∃x append(cons(a1,cons(a2,cons(a3,cons(a4,cons(a5,cons(a6,cons(a7,nil)))))))), nil,x)
[25380]

2.4 ∀x append(nil,x,x) &
∀x∀y∀z∀w(append(y,z,w) ⊃ append(cons(x,y),z,cons(x,w)))
⊃
∃x append(cons(a1,cons(a2,cons(a3,cons(a4,cons(a5,cons(a6,cons(a7,cons(a8, nil)))))))),
nil,x) [292020]

Comment: 2.1--2.4 become very much easier if ∀x append(nil,x,x) is annotated as ∀1x append(nil,x,x), i.e. the first premiss is given contraction depth 1 (e.g. 2.4 executes in 4890 instead of 292020).

3. Problems 39--43 from Pelletier's collection.

3.1 ¬ ∃x∀y(member(y,x) ↔ ¬ member(x,x)) [10]

3.2 ∃y∀x(member(x,y) ↔ member(x,x))
⊃
¬ ∀x∃y∀z(member(z,y) ↔ ¬ member(z,x)) [270]

3.3 ∀z∃y∀x(member(x,y) ↔ member(x,z) & ¬ member(x,x))
⊃ ¬ ∃x∀ymember(y,x) [40]

3.4 ¬ ∃x∀y(member (y,x) ↔ ¬ ∃w(member(x,w) & member(w,x))) [10]

3.5 ∀x∀y(equal(x,y) ↔ ∀z(member (z,x) ↔ member(z,y)))
⊃
∀x∀y(equal(x,y) ⊃ equal(y,x)) [10240]

4. Existence

4.1 ∀x(p(x) ⊃ p(h(x))∨p(g(x))) & ∃xp(x)&
∀x¬ p(h(x)) ⊃ ∃xp(g(g(g(g(g(x)))))) [43670]

4.2 ∀x(p(x) ⊃ p(h(x))∨p(g(x)))& ∃xp(x) &
∀x¬ p(h(x)) ⊃ ∃xp(g(g(g(g(g(g(x))))))) [49310]

4.3 ∀x(p(x) ⊃ p(h(x))∨p(g(x))) & ∃xp(x) & ∀x¬ p(h(x)) ⊃
∃xp(g(g(g(g(g(g(g(x)))))))) [57750]

Unify

5.1 ∀x0∃x1∃x2∃x3∃x4∃x5(p(x1,x2,x3,x4,x5) ↔
p(f(x0,x0),f(x1,x1),f(x2,x2),f(x3,x3),f(x4,x4))) [20]

5.2 ∀x0∃x1∃x2∃x3∃x4∃x5∃x6 ∃x7∃x8∃x9∃x10(
p(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10) ↔
p(f(x0,x0),f(x1,x1),f(x2,x2),f(x3,x3),f(x4,x4),f(x5,x5),
f(x6,x6),f(x7,x7),f(x8,x8),f(x9,x9))) [110]

5.3 ∀x0∃x1∃x2∃x3∃x4∃x5 ∃x6∃x7∃x8∃x9∃x10∃x11 ∃x12∃x13∃x14∃x15(
p(x1,x2,x3,x4,x5,x6,x7, x8,x9,x10,x11,x12,x13,x14,x15) ↔
p(f(x0,x0),f(x1,x1),f(x2,x2),f(x3,x3), f(x4,x4),
f(x5,x5),f(x6,x6),f(x7,x7),f(x8,x8),f(x9,x9),
f(x10,x10),f(x11,x11), f(x12,x12),f(x13,x13),f(x14,x14))) [3190]

Comment: 5.1--5.3 illustrate an exponential series (in the present algorithm) due to the parameter check and occur check.

6. Simple

6.1 ∀x∀y(p(x) ⊃ q(y)) ↔ (∃xp(x) ⊃ ∀yq(y)) [10]

6.2 ¬ ∃x1∃x2∃x3∃x4∃x5∃x6 ∃x7∃x8∃x9∃x10p(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)
↔
∀x1∀x2∀x3∀x4∀x5∀x6∀x7 ∀x8∀x9∀x10¬ p(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10) [20]

6.3 ∃x(p(x) & ∀y(q(y) ⊃ r(x,y))) &
¬ ∃x(q(x) & ∀y(p(y) ⊃ r(x,y))) ⊃ ¬ ∀x(p(x) ⊃ q(x)) [20]

6.4 ∀x(p(x) ↔ ∃y(q(y) & r(x,y))) → ∀x(¬ p(x) ↔ ∀y ¬ (q(y) & r(x,y))) [10]

6.5 ∀x(p(x) ↔ q(x)) &
∀x(r(x)↔ s(x)) ⊃ ∀x(p(x) & ¬ r(x) ↔ q(x) & ¬ s(x)) [20]

6.6 ∀x(p(x) ⊃ q(x)) & ∀x(q(x) ⊃ s(x)) ⊃ ∀x(¬ ¬ p(x) ⊃ ¬ ¬ s(x)) [10]

6.7 ∃xp(x) & ∀x∀y(p(y) & q(x) ⊃ ¬ r(x)) ⊃ ∀x(q(x) & r(x) ⊃ s(x)) [10]

6.8 ∀x(p(x) ⊃ ∃y(q(x,y) & r(y))) & ∃x∃y(q(x,y)∨p(x)) ⊃ ∃x∃yq(x,y) [10]

6.9 ∀x(p(x) ⊃ r(x)∨∃yq(x,y)) &
∀x(r(x) ⊃ ¬ ∃xp(x)) & ∃xp(x) ⊃ ∃x∃yq(x,y) [10]

6.10 ∃x(p(x) & ∀y(q(y) ⊃ r(x,y)∨r(y,x))) &
∃x(q(x) & ∀y(p(y) ⊃ ¬ r(x,y))) ⊃
∃x∃y(p(x) & q(y) & r(x,y)) [10]

6.11 ¬ ∃x∀y(q(y) ⊃ r(x,y)) & ∃x∀y(s(y) ⊃ r(x,y)) ⊃ ¬ ∀x(q(x) ⊃ s(x)) [10]

6.12 p(a) & ¬ p(f(f(f(f(f(a)))))) ⊃ ¬ ¬ ∃x(p(x) & ¬ p(f(x))) [150]

6.13 ∀x(p(x) ↔ q(x)∨r(x)∨∃ys(x,y)) & ∃x∃y(s(y,x)∨g(x)) &
∀x(g(x) ↔ ∃ys(x,y) ∨∃z(r(z)∨q(z)∨s(a,z))) ⊃
∃xq(x)∨∃xr(x)∨∃x∃ys(x,y) [270]

6.14 ∀x1∀x2∃y1∃y2(p(x1) & q(x2) & r(y1) & s(y2)) ↔
∃y1∃y2∀x1∀x2(p(x1) & q(x2) & r(y1) & s(y2)) [30]

6.15 ∀x(p(x) ⊃ ¬ ∃y(q(y) & r(x,y))) & ∀x(t(x)⊃ ∃y(s(y) & r(x,y))) & ∀x(p(x) ⊃
¬ ¬ t(x)) & ∀y(s(y) ⊃ q(y)) ⊃ ¬ ∃xp(x) [160]

Comment:the designation 'simple' is here pretty arbitrary: these are just some short formulae, written down more or less at random, that happen to be easily proved by the present algorithm. Other short formulae take 'forever'.

7. Problematic

q(a1,a2,a1,a2) ⊃
∃x1∃x2∃y1∃y2((p(x1) & p(x2) ↔p(y1) & p(y2)) & q(x1,x2,y1,y2)). [20]

q(a1,a2,a3,a1,a2,a3) ⊃ ∃x1∃x2∃x3∃y1∃ y2∃y3((p(x1) & p(x2) & p(x3) ↔
p(y1) & p(y2) & p(y3)) & q(x1,x2,x3,y1,y2,y3)). [210]

q(a1,a2,a3,a4,a1,a2,a3,a4) ⊃ ∃x1∃x2 ∃x3∃x4∃y1∃y2∃y3∃y4(
(p(x1) & p(x2) & p(x3) & p(x4)↔p(y1) & p(y2) & p(y3) & p(y4)) &
q(x1,x2,x3,x4,y1,y2,y3,y4)) [16680]

q(a1,a2,a3,a4,a5,a1,a2,a3,a4,a5)⊃
∃x1∃x2∃x3∃x4∃x5∃y1∃y2∃y3∃y4∃y5((p(x1) & p(x2) & p(x3) & p(x4) & p(x5)↔
p(y1) & p(y2) & p(y3) & p(y4) & p(y5)) & q(x1,x2,x3,x4,x5,y1,y2,y3,y4,y5)). [hopeless]

Comment: for the present algorithm, this quickly becomes hopeless and remains so as long as the left premiss is proved first in ⊃ &.

Fruit & Cheese This last example shows the output of the algorithm in response to the question A&B,..⊃ food(X) and A&B,.. ⊃ ¬ ¬ food(X) respectively at contraction level 111, i.e. equivalences, implications, and universal formulae are used only at contraction level 1. '..solutions skipped' reports top level sifting of answers.

8.1

∀x(¬ fruit(x) ⊃ ¬ apple(x) & ¬ pear(x))
&

8.

```
∀x(fruit(x)∨bread(x)∨cheese(x)∨whiskey(x)⊃ food(x))
&
pear(moltke) & apple(grannysmith) & apple(reddelicious) & cheese(stilton)
&
(apple(juicyfruit)∨pear(juicyfruit))
&
(bread(rye)∨whiskey(rye))
&
(¬ cheese(brie)⊃ fruit(brie))
⊃
food(X).
1 1 1
X=rye
[20]
More?(y/n) y

X=stilton
[80]
More?(y/n) y

1 solution skipped
no
[90]
```

8.2

```
∀x(¬ fruit(x)⊃ ¬ apple(x) & ¬ pear(x))
&
∀x(fruit(x)∨bread(x)∨cheese(x)∨whiskey(x)⊃ food(x))
&
pear(moltke) & apple(grannysmith) & apple(reddelicious) & cheese(stilton)
&
(apple(juicyfruit)∨pear(juicyfruit))
&
(bread(rye)∨whiskey(rye))
&
(¬ cheese(brie)⊃ fruit(brie))
⊃
¬ ¬ food(X).
1 1 1
X=rye
[40]
More?(y/n) y

40 solutions skipped
X=stilton
[300]
More?(y/n) y

36 solutions skipped
X=moltke
[340]
More?(y/n) y

6 solutions skipped
X=juicyfruit
[340]
```

```
More?(y/n) y

6 solutions skipped
X=reddelicious
[330]
More?(y/n) y

6 solutions skipped
X=grannysmith
[380]
More?(y/n) y

6 solutions skipped
X=brie
[740]
More?(y/n) y

89 solutions skipped
no
[1960]
```

Received 26 March 1991