A PROLOG Interpreter for First-Order Intuitionistic Logic
(Extended Abstract)

L. Thorne McCarty and Leon A. Shklar
Computer Science Department
Rutgers University
New Brunswick, NJ 08903, USA
mccarty@cs.rutgers.edu

March 1994

Abstract
What does it mean to write an “efficient” theorem prover for first-order intuitionistic logic? This paper suggests a possible answer to the question, following the philosophy of logic programming. We decompose intuitionistic logic into two syntactically restricted subsets: (i) the class of simple embedded implications and (ii) the class of disjunctive and existential assertions. Each of these subsets has been studied previously as a possible logic programming language, and we show how to write a simple interpreter (in PROLOG) for each one. We then combine the two interpreters into a single PROLOG interpreter for full first-order intuitionistic logic, and analyze its performance on several benchmark examples.

Key Words and Phrases: intuitionistic logic programming; disjunctive logic programming; interpreters for extended logic programming languages; theorem-provers for nonclassical logics.

Presented at: ILPS ’94 Poster Session

Copyright © 1994 by L.T. McCarty and L.A. Shklar
1 Introduction

There have been numerous attempts to develop intuitionistic theorem-provers in recent years. For example, Fitting [7, 6] describes a straightforward translation of his tableau proof procedure for intuitionistic logic (and several modal logics) [5] into PROLOG, and Felty and Miller [3] describe the translation of a sequent calculus proof procedure for intuitionistic logic (and also classical logic) into λ-PROLOG. Proof search can be very inefficient in such systems, however, and subsequent work has attempted to improve the efficiency of intuitionistic theorem-proving. Thus Beeson [1] shows how a PROLOG-style search heuristic can be applied to an extension of Gentzen’s system G3, and Shankar [29] shows how a generalized form of skolemization can be used to enforce the eigenvariable condition in G3 and related systems, thus eliminating a substantial amount of unnecessary search. Wallen’s work on theorem-proving for modal logics [32] also has an intuitionistic version, based on a translation into S4. For additional related work, see [26, 27, 10, 2, 28, 24]

Unfortunately, it is not entirely clear what “efficiency” means in this context. The validity problem for intuitionistic logic is PSPACE complete in the propositional case [30], and thus it is easy to construct examples that would confound the search heuristics of any conceivable theorem-prover (unless, of course, the polynomial hierarchy collapses and P=NP). Moreover, since first-order intuitionistic logic is undecidable, it can be used to encode problems that are complete for exponential time, complete for double-exponential time, . . . , and so on. Faced with this potentially explosive complexity, what should one do?

Our approach in this paper is based on the philosophy of logic programming. Horn clause logic (with function symbols) is also undecidable, but since it is a syntactically restricted subset of first-order classical logic, with a very simple proof procedure, the authors of logic programs and the designers of deductive databases are able to express their ideas in a form that is, in practice, efficiently computable. In this paper, we extend the class of Horn clauses (without function symbols) to two orthogonal subsets of first-order intuitionistic logic: (i) the class of simple embedded implications; and (ii) the class of disjunctive and existential assertions. Each of these subsets has been studied independently as a possible logic programming language, the first subset in [9, 8, 16, 17, 25, 22] and the second subset (essentially) in [23, 15]. We show in Section 2 that the union of these subsets is equivalent to full first-order intuitionistic logic, and that a sound and complete proof theory for this syntactically restricted language can be formalized in a radically simplified sequent calculus. Section 2 thus provides the foundation for an interpreter for first-order intuitionistic logic.

In Section 3, we describe an implementation of this interpreter in PROLOG. More precisely, we describe three interpreters: one for each of the two subsets and a third that subsumes the first two. We use vanilla PROLOG code to explain how each of these interpreters works, and we analyze the performance of the actual code on several benchmark examples. One observation that emerges from this analysis is that the performance of our interpreter depends critically on which subsets of the language are used to encode which kinds of problems. This is not a surprising observation, in retrospect, but it justifies our initial decision to develop a specialized theorem-prover for a syntactically restricted language.

2 Intuitionistic Logic in Clausal Form

Intuitionistic logic, like most logics, can be defined in two ways: model-theoretically or proof-theoretically. Since we plan to develop an unconventional proof theory for a syntactically restricted
language, it makes sense to begin with a neutral model theory. We must then show: (i) that the
unconventional proof theory is sound and complete for the syntactically restricted language; and
(ii) that the syntactically restricted language is expressively equivalent to the full language.

We assume that the reader is generally familiar with the Kripke semantics for intuitionistic logic
[13, 4], and we simply review our notation here. Let $L$ be a function-free first-order language, and
let $L(c)$ be the language $L$ augmented by an arbitrary set of new constants $c$. We write a Kripke
structure for $L$ as a quadruple $(K, \leq, h, u)$, where $K$ is a nonempty set of states, `$\leq$' is a partial
order on $K$, $h$ is a monotonic mapping from the states of $K$ to sets of ground atomic formulae in
$L(c)$, and $u$ is a monotonic mapping from the states of $K$ to nonempty sets of individual constants
in $L(c)$, with $h(s)$ always restricted to formulae in $L(u(s))$. For a ground atomic formula $A$, we
say that "$s$ forces $A$ in $K$" and write $s, K \models A$ if and only if $A \in h(s)$. The definition of forcing
for conjunction, disjunction and existential quantification depends only on a single state, $s$, as in
classical logic, but the definitions for implication and universal quantification are nonclassical:

$$s, K \models B \iff s', K \models A \text{ implies } s', K \models B \text{ for every } s' \geq s \text{ in } K,$$

and for all constants $c$ in $u(s')$.

If $s, K \models A$ for every $s \in K$, we say that $(K, \leq, h, u)$ satisfies $A$. If $s, K \models A$ for every $s \in K$ such
that the individual constants in $A$ are in $u(s)$, then we say that $A$ is true in $(K, \leq, h, u)$. Finally, if
$\Phi$ is a set of sentences and $\psi$ is a sentence, we write $\Phi \models \psi$ if and only if $\psi$ is true in every Kripke
structure that satisfies $\Phi$.

For our syntactically restricted language, we are interested in two main classes of formulae. The
first is the class of simple embedded implications:

$$P(x) \leftarrow A_1(x) \land A_2(x) \land \cdots \land A_n(x), \quad n \geq 0,$$

in which $P(x)$ is an atomic formula and each $A_i(x)$ is either an atomic formula, $R_i(x)$, or an
embedded implication $(\forall y)[R_i(x; y) \leftarrow \bigwedge_j Q_{ij}(x; y)]$ with (zero or more) embedded universal quan-
tifiers. Note that $R_i(x; y)$ and $Q_{ij}(x; y)$ are also atomic formulae in this definition. Thus a simple
embedded implication is a Horn clause if every $A_i(x)$ is atomic, but, in general, the $A_i(x)$ are
themselves (definite) Horn clauses. The second class of formulae is the class of disjunctive and
existential assertions:

$$P(x) \Rightarrow \bigvee_{i=1}^n Q_i(x), \quad \text{and} \quad P(x) \Rightarrow (\exists y) \bigwedge_{j=1}^m Q_j(x; y),$$

in which the $x$ variables are universally quantified at the top level, and no $x$ variables are allowed
on the right-hand side that do not also appear in the atomic formula, $P(x)$, on the left-hand side.
Again, $Q_i(x)$ and $Q_j(x; y)$ are atomic formulae in these definitions.

Now let $R$ be a set of simple embedded implications plus disjunctive and existential assertions,
called rules. We adopt the following simplified sequent calculus:

**Definition 2.1:**

1. There is a rule $P(x) \leftarrow \bigwedge_{i=1}^n A_i(x)$ in $R$ and a ground substitution
   $\theta$ such that $A = P(x)\theta$ and
   $$R \vdash A_i(x)\theta, \quad \text{for } i = 1, \ldots, n.$$
(I2) \( R \vdash A \iff \bigwedge_{i=1}^{n} Q_i \) if \( R \cup \bigcup_{i=1}^{n} \{Q_i\} \vdash A \)

(I3) \( R \vdash (\forall y)A(y) \) if \( R \vdash A(c) \) for some tuple of constants \( c = \langle c_1, c_2, \ldots, c_k \rangle \) that do not appear in \( R \) or \( A(y) \).

(I4) \( R \vdash A \) if there is a rule \( P(x) \Rightarrow \bigvee_{i=1}^{n} Q_i(x) \) in \( R \) and a ground substitution \( \theta \) such that \( R \vdash P(x)\theta \) and

\[
R \cup \{Q_i(x)\theta\} \vdash A, \quad \text{for } i = 1, \ldots, n.
\]

(I5) \( R \vdash A \) if there is a rule \( P(x) \Rightarrow (\exists y) \bigwedge_{j=1}^{m} Q_j(x; y) \) in \( R \) and a ground substitution \( \theta \) such that \( R \vdash P(x)\theta \) and

\[
R \cup \bigcup_{j=1}^{m} \{Q_j(x; c)\theta\} \vdash A
\]

for some tuple of constants \( c = \langle c_1, c_2, \ldots, c_k \rangle \) that do not appear in \( R \) or \( A \).

Suppose \( \psi = (\forall y)[R(y)\iff \bigwedge_{i=1}^{n} Q_i(y)] \) is a universally quantified implication, called a goal. We write \( R \vdash \psi \) if there exists a finite tree of sequents with \( R \vdash \psi \) at the root and with each edge justified by one of the inferences (I1)–(I5) from Definition 2.1. As special cases, we can also take \( \psi = R \iff \bigwedge_{i=1}^{n} Q_i \) and \( \psi = R \). Note that the terminal nodes of a proof tree can only be justified by the inference (I1) and a rule \( P(x) \iff \) with a null antecedent.

The language we have discussed so far is actually positive intuitionistic logic. To represent negation, we add a special nullary predicate ‘\( \bot \)’ to \( L \) to denote a contradiction, and we write \( \neg A \) as an abbreviation for \( \bot \iff A \). Thus \( \neg R \) and \( P \iff \neg Q \) are within the class of simple embedded implications. We now have two choices: If we do not add any special rules for ‘\( \bot \)’, then we have minimal logic [12]. For intuitionistic logic, we need to add the condition ‘\( \forall s \in K : \bot \not\in s \)’ to the model theory, and we need to add the inference ‘\( R \vdash A \) if \( R \vdash \bot \)’ to the proof theory [11]. In the present paper, we follow the minimal logic approach. It is a simple matter to modify our interpreter so that it computes intuitionistic negation, but this will often lead to an inefficient proof search [17]. The examples in the paper that use negation will work under either interpretation.

**Example 2.2:** This is an elaboration of an example suggested by Gabbay and Reyle [10]. Let \( R \) consist of the following Horn clauses:

\[
\begin{align*}
\text{Parent}(x, y) & \iff \text{Mother}(x, y) \quad (1) \\
\text{Parent}(x, y) & \iff \text{Father}(x, y) \quad (2) \\
\text{GrandParent}(x, y) & \iff \text{Parent}(x, z) \land \text{Parent}(z, y) \quad (3) \\
\bot & \iff \text{GrandParent}(x, y) \quad (4)
\end{align*}
\]

The interpretation of rules (1)–(3) is obvious, and rule (4) tells us: “There are no grandparents”. Now consider the query:

\[
(\exists v)(\forall u)[\neg \text{Mother}(u, v) \land \neg \text{Father}(u, v)]? \quad (5)
\]
To express this query, we note that universal quantification distributes over conjunction in intuitionistic logic, and we can therefore write the following simple embedded implication as an auxiliary rule:

\[ Q_1 \Leftarrow (\forall u)[\bot \Leftarrow \text{Mother}(u, v)] \land (\forall u)[\bot \Leftarrow \text{Father}(u, v)]. \]  

(6)

Suppose the set \( Q \) contains auxiliary rules, such as (6). We can now express query (5) by asking: \( R \cup Q \models Q_1 \) ? The Kripke model in Figure 1 shows that the answer is: No. The reader should be able to verify that rules (1)–(4) and (6) are satisfied at \( s_1, s_2 \) and \( s_3 \) in this model, but \( 'Q_1' \) is not forced at \( s_1 \). Figure 1 also shows that we could replace (6) by:

\[ Q_1 \Leftarrow (\forall u)[\bot \Leftarrow \text{Mother}(u, v)], \]  

(7)

\[ Q_1 \Leftarrow (\forall u)[\bot \Leftarrow \text{Father}(u, v)], \]  

(8)

and the query \( 'Q_1' \) would still fail.

Example 2.3: Now let \( R \) consist of rules (1)–(4), but consider the query:

\[ \neg (\forall v)(\exists u)[\text{Mother}(u, v) \lor \text{Father}(u, v)]? \]  

(9)

Notice that (9) is equivalent to (5) in classical logic, but not in intuitionistic logic. If we tried to rewrite (9) in the same way we rewrote (5), we would not have a legal expression in our syntactically restricted language. However, we can construct a query equivalent to (9) by introducing new predicates \( 'D_0', 'D_1' \) and \( 'D_2' \), as follows:

\[ Q_2 \Leftarrow [\bot \Leftarrow D_0] \]  

(10)

\[ D_1(v) \Leftarrow D_0 \]  

(11)

\[ D_1(v) \Rightarrow (\exists u)D_2(u, v) \]  

(12)

\[ D_2(u, v) \Rightarrow \text{Mother}(u, v) \lor \text{Father}(u, v) \]  

(13)

Since (10)–(13) are legal expressions, we can add them to the set \( Q \). We can now express query (9) by asking: \( R \cup Q \models Q_2 \) ? Figure 2 shows that the answer is: Yes. We have drawn the proof tree with the initial goal at the top, using small numerals to indicate which inferences from Definition 2.1 are used at each step. Since the formulae on the left-hand side of a sequent are elements of a set, we have rearranged them freely to highlight the closed branches. It is obvious that the four bottom branches would succeed, using rules (1)–(3) and several applications of the inference (I1), and we have omitted the details.

The justification for our approach lies in the following theorems, which are proven in the full version of this paper. First, we need to show that our syntactically restricted language is equivalent to full first-order intuitionistic logic:
\[\vdash Q_2\]
\[\vdash \bot \iff D_0\]
\[D_0 \vdash \bot\]

\[\vdash D_1(a)\]
\[\vdash D_0, D_2(b, a) \vdash \bot\]
\[D_0, D_2(b, a) \vdash D_1(b)\]
\[D_0, D_2(b, a), D_2(c, b) \vdash \bot\]
\[\vdash D_0, D_2(b, a) \vdash D_0\]
\[\vdots, D_2(c, b) \vdash D_2(c, b)\]
\[\ldots, F(c, b) \vdash \bot\]
\[\ldots, D_2(b, a) \vdash D_2(b, a)\]
\[\ldots, F(c, b), M(b, a) \vdash \bot\]
\[\ldots, F(c, b) \vdash \bot\]
\[\ldots, M(b, a) \vdash \bot\]
\[\ldots, M(c, b) \vdash \bot\]
\[\ldots, M(c, b), F(b, a) \vdash \bot\]
\[\ldots, G(c, a) \vdash \bot\]
\[\ldots, G(c, a) \vdash \bot\]
\[\ldots, G(c, a) \vdash \bot\]
\[\ldots, G(c, a) \vdash \bot\]

Figure 2: A Proof Tree for ‘Q_2’.

**Theorem 2.4:** Let \(\phi\) be any sentence in \(\mathcal{L}\). There exists a translation \(Tr(\phi)\) into a set of simple embedded implications plus disjunctive and existential assertions, in an extended language \(\mathcal{L}' \supseteq \mathcal{L}\), such that \(\{\phi\} \models \psi\) if and only if \(Tr(\phi) \models \psi\) for every sentence \(\psi \in \mathcal{L}\). \(\square\)

Second, we need to show that the simplified sequent calculus in Definition 2.1 is sound and complete for the syntactically restricted language:

**Theorem 2.5:** Let \(\mathcal{R}\) be a set of simple embedded implications plus disjunctive and existential assertions, called *rules*, and let \(\psi\) be a closed Horn clause called a *goal*. Then: \(\mathcal{R} \models \psi\) if and only if \(\mathcal{R} \vdash \psi\). \(\square\)

Finally, in order to use Definition 2.1 in a PROLOG-style interpreter, we need to eliminate the explicit existential quantifiers. To see how to do this, notice that we can systematically replace the constant ‘b’ in Figure 2 with ‘sk(a)’, and then systematically replace the constant ‘c’ with ‘sk(sk(a))’. Thus, we could replace rule (12) with the skolemized variant: \(D_1(v) \Rightarrow D_2(sk(v), v)\), and the structure of the proof would remain the same. (Notice that we are introducing skolem functions only in this limited context, and not adding arbitrary function symbols to the language \(\mathcal{L}\) itself.) In general, we have:

**Theorem 2.6:** Let \(\mathcal{R}\) be a set of simple embedded implications plus disjunctive and existential assertions, and let \(\mathcal{R}'\) be the same as \(\mathcal{R}\) but with the existential assertions replaced by their skolemized variants. Then: \(\mathcal{R} \models \psi\) if and only if \(\mathcal{R}' \vdash \psi\).
With this reduction, we need only write an interpreter to handle embedded implications and disjunctive assertions, alone, and in combination — a task to which we now turn.

3 A PROLOG Interpreter

Our complete interpreter for intuitionistic logic is relatively complex, and we develop it here in stages. Section 3.2 describes our interpreter for embedded implications, and Section 3.3 describes our interpreter for disjunctive assertions. Since these partial interpreters may be of independent interest, we will show how they work on several simple examples. We will then combine the two interpreters, in Section 3.4, to obtain coverage of the full language.

Since the two interpreters are based on different control structures — SLD and OLD, respectively — we begin by comparing these two approaches in the simple case of Horn clause logic.

3.1 Horn Clauses: SLD vs. OLD

Figure 3 shows the familiar three-line PROLOG interpreter for SLD resolution. Figure 4 is based on Warren’s implementation [33] of OLD resolution [31].

\begin{verbatim}
  sld(true) :- !.
  sld((Goal1,Goal2)) :- !,sld(Goal1),sld(Goal2).
  sld(Goal) :- clause(Goal,Body),sld(Body).
\end{verbatim}

Figure 3: Proof by SLD.

\begin{verbatim}
  old([Ans<-[]],Stack) :- write(Ans),nl,!,extend(Stack).
  old([Ans<-[],Frame|Node],Stack) :-
    copy_term(Frame,A<-[Call|Goals]), Call=Ans,
    !,old([A<-Goals|Node],Stack).
  old(Node,Stack) :- Node = [_[<-[Goal|_]|_],
    findall(Goal<-Body,rule(Goal,Body),Frames),
    extend([branch(Frames,Node)|Stack]).
  extend([branch([],_)|Stack]) :- !,extend(Stack).
  extend([branch([Frame],Node)|Stack]) :- !,old([Frame|Node],Stack).
  extend([branch([Frame|Frames],Node)|Stack]) :-
    old([Frame|Node],[branch(Frames,Node)|Stack]).
\end{verbatim}

Figure 4: Proof by OLD.

In the OLD version, backtracking information is maintained explicitly in the variable \texttt{Stack}. \texttt{Stack} is a list of branches in the form \texttt{branch(Frames,Node)}, where \texttt{Frames} and \texttt{Node} are both lists of frames. A frame has the form \texttt{Ans<-Goals}, where \texttt{Ans} is an atom and \texttt{Goals} is a list of atoms. The call to \texttt{findall} in the third clause of \texttt{old} collects a list of frames whose \texttt{Ans} atom would unify with the first goal — the calling goal — in the first frame of the current \texttt{Node}. These frames are then packaged into a new branch, which is pushed onto the current \texttt{Stack} and processed by \texttt{extend}. When the list of goals is empty, the second clause of \texttt{old} unifies \texttt{Ans} with \texttt{Call} and begins processing the next frame in the current \texttt{Node}, minus the calling goal. Finally, if the list of unprocessed frames is empty, the first clause of \texttt{old} writes out the answer and calls \texttt{extend}. 

3 A PROLOG INTERPRETER

to process the stack again. Notice that old itself is deterministic, and that it writes out all the solutions to the initial goal before failing with an empty stack.

Warren’s version of OLD differs from Figure 4 by maintaining the information in Stack implicitly in PROLOG, but it is otherwise identical. Warren uses this simple code as the foundation of a memoing interpreter, OLDT, which is guaranteed to terminate on function-free logic programs [34]. In contrast, we use the explicit Stack variable in the first clause of old to “restart” the interpreter when it encounters disjunctive assertions, as we will see in Section 3.3. But it is also possible to use this information to “memoize” the search for intuitionistic proofs, a topic that is beyond the scope of the present paper.

3.2 Embedded Implications

The fragment of intuitionistic logic consisting entirely of embedded implications forms a natural logic programming language, as several researchers have noted [9, 8, 16, 17, 25, 22]. We can thus write an interpreter for embedded implications simply by extending the SLD interpreter in Figure 3. Figure 5 shows one way to do this.

provel(true,_,_,_) :- !.
provel((G1,G2),HypDB,Vars,N) :- !,provel(G1,HypDB,Vars,N),provel(G2,HypDB,Vars,N).
provel((Goal:-Hyp),HypDB,Vars,N) :-
    append(Hyp,HypDB,NewHypDB),N1 is N+1,
    add_variables((Goal:-Hyp),N,Vars,Vars1),
    !,provel(Goal,NewHypDB,Vars1,N1).
provel(forall(Ys,(Goal:-Hyp)),HypDB,Vars,N) :-
    append(Hyp,HypDB,NewHypDB),N1 is N+1,
    generate_constants(Ys,N1),
    add_variables((Goal:-Hyp),N,Vars,Vars1),
    !,provel(Goal,NewHypDB,Vars1,N1).
provel(Goal,HypDB,Vars,_) :- member(Goal,HypDB),\+violation(Vars).
provel(Goal,HypDB,Vars,N) :- clause(Goal,Body),\+violation(Vars),
    provel(Body,HypDB,Vars,N).

Figure 5: An Interpreter for Embedded Implications.

The first, second and sixth clauses of provel are identical to the three clauses of sld, except for the presence of additional arguments. The third clause handles the case of an embedded implication without embedded universal quantifiers. The idea here is simple, and corresponds exactly to the inference (I2) in Definition 2.1. To prove that Goal follows from Hyp, we append Hyp to the current HypDB and try to prove that Goal follows from NewHypDB. Since Hyp and HypDB are each assumed to be a list of atoms, this test can be implemented easily by the member predicate in the fifth clause. Although HypDB may contain a number of uninstantiated variables, arising from the implication (Goal:-Hyp), an analysis of Definition 2.1 shows that this causes no problems. In fact, the unification of these uninstantiated variables is exactly what we want.

The other additional arguments in Figure 5, Vars and N, are not actually needed unless our rulebase includes embedded implications with embedded universal quantifiers. This case is handled by the fourth clause of provel. The problem here is to enforce the eigenvariable condition in inference (I3) of Definition 2.1. There seem to be two natural approaches to this problem: (i) We can generate skolem functions dynamically at the time we encounter the universally quantified goal.
This is the approach adopted in [10, 29]. (ii) We can index the constants and the variables by noting the level of the proof at which they were created. This is the approach adopted in [17, 24]. Using the terminology suggested in [17], we imagine that the proof begins in an initial tableau at level 0, and that each attempt to prove an embedded implication creates an auxiliary tableau with its level incremented by 1. In the move from level \( n \) to level \( n + 1 \), all of the new uninstantiated variables occurring in the embedded implication are considered to be variables of type \(?x\) and assigned the index \( n \), while all of the universally quantified variables occurring in the embedded implication are considered to be constants of type \(!y\) and assigned the index \( n + 1 \). Subsequently, unification fails whenever a variable of type \(?x\) and index \( i \) is bound to a term containing a constant of type \(!y\) and index \( j \), for \( i < j \).

The interpreter in Figure 5 adopts this second approach. \( \text{Vars} \) is a list of \(?x\) variables plus their indices, and new variables with index \( N \) are added to this list by the predicate \( \text{add\_variables} \) in the third and fourth clauses. In the fourth clause, \( \text{Ys} \) is a list of the universally quantified variables that appear in the embedded implication. (Note the assumptions made here about the syntax of the rulebase.) These variables are instantiated by \( \text{generate\_constants} \) to unique ground terms with the index \( N1 \). Finally, in the fifth and sixth clauses of \( \text{prove1} \), the predicate \( \text{violation} \) tests the eigenvariable condition.

Let us see how \( \text{prove1} \) works with Example 2.2. Consider the query ‘\( Q_1 \)’ as defined by (7)–(8). Assume that (7) is written as follows:

\[
q_1 :- \text{forall}([U],(f:-[\text{mother}(U,V)]))
\]

The proof begins at level 0, but immediately adds \( \text{mother}(y(n1,1),V) \) to the database at level 1 and tries to prove \( f \). Note that \( y(n1,1) \) has the index 1 and \( V \) is assigned the index 0. The only way to prove \( f \) is to unify both \( \text{mother}(X,Z) \) and \( \text{mother}(Z,Y) \) with \( \text{mother}(y(n1,1),V) \), but this violates the eigenvariable condition. The same violation occurs when we try to unify \( \text{father}(X,Z) \) and \( \text{father}(Z,Y) \) with \( \text{father}(y(n2,1),V) \). Thus the query fails.

In the “no grandparents” example, it is easy to understand the search for a proof by an inspection of the declarative rulebase, and this is one of the reasons that embedded implications make a useful logic programming language. However, it is also possible to construct examples in which the proof search is impossible to control.

Example 3.1: We will use this example as a benchmark both here and in the following section.
Let \( \mathcal{R} \) consist of the following rules:

\[
\begin{align*}
\text{NotColorable} & \iff \exists \exists \text{Colorable} \\
\perp & \iff \text{Colorable} \land \text{Node}(x) \land \text{Node}(y) \land \text{Edge}(x, y) \land \text{SameColor}(x, y) \\
\text{SameColor}(x, y) & \iff \text{Red}(x) \land \text{Red}(y) \\
\text{SameColor}(x, y) & \iff \text{Green}(x) \land \text{Green}(y) \\
\text{SameColor}(x, y) & \iff \text{Blue}(x) \land \text{Blue}(y) \\
\text{Red}(x) & \iff \text{Node}(x) \land [\perp \iff \text{Green}(x)] \land [\perp \iff \text{Blue}(x)] \\
\text{Green}(x) & \iff \text{Node}(x) \land [\perp \iff \text{Blue}(x)] \land [\perp \iff \text{Red}(x)] \\
\text{Blue}(x) & \iff \text{Node}(x) \land [\perp \iff \text{Red}(x)] \land [\perp \iff \text{Green}(x)]
\end{align*}
\]

Intuitively, rule (14) says that a graph is ‘NotColorable’ if the assumption that it is ‘Colorable’ leads to a contradiction. Rules (15)–(18) tell us that it is contradictory for a graph to be ‘Colorable’ and have adjacent nodes of the ‘SameColor’. Rule (19) tells us that a node is ‘Red’ if it is not ‘Green’ and not ‘Blue’, and similarly for rules (20) and (21). Assume that a particular graph is
represented by sets of ground atoms $\mathcal{N}$ for the nodes and $\mathcal{E}$ for the edges. Then the graph is not colorable if and only if $\mathcal{R} \cup \mathcal{N} \cup \mathcal{E} \models \text{NotColorable}$.

The performance of \texttt{prove1} on this example is very poor, even for small graphs. We tried it on a clique of three nodes (using Quintus PROLOG on a Sun 380), but aborted the proof search after several days of computation. We estimate that the interpreter would have taken more than $5.46 \times 10^{68}$ CPU hours to discover that the proof of ‘NotColorable’ fails in this case! There are two main problems: (1) Given our encoding of graph-colorability, \texttt{prove1} generates an exponential number of hypothetical databases that are intuitively inconsistent, e.g., databases in which both \texttt{red(a)} and \texttt{green(a)} are true. This is a problem with the logic, not a problem with the proof procedure, and there seems to be no solution using a monotonic form of negation, since the addition of a constraint such as $\bot \Leftarrow \text{Red}(x) \land \text{Green}(x)$ just enlarges the search space and makes the problem worse. (There are possible solutions using negation-as-failure, but this is beyond the scope of the present paper.) (2) A problem with the interpreter itself is the fact that it sequentializes the proof of the two goals, $G_1$ and $G_2$, even when they can be shown to succeed or fail independently. A quick fix for this problem is to insert a \texttt{cut} between the proof of $G_1$ and the proof of $G_2$ whenever their variables are disjoint. With this modification, and using cliques of size $n$ as a test case, we have the following results:

<table>
<thead>
<tr>
<th>nodes</th>
<th>result</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>fails</td>
<td>35 min 14 sec</td>
</tr>
<tr>
<td>4</td>
<td>succeeds</td>
<td>2 min 12 sec</td>
</tr>
<tr>
<td>5</td>
<td>succeeds</td>
<td>18 min 28 sec</td>
</tr>
<tr>
<td>6</td>
<td>succeeds</td>
<td>206 min 18 sec</td>
</tr>
</tbody>
</table>

This is better, but still not very good. These results suggest that graph-colorability (and similar problems) should not be encoded using embedded implications. We will analyze a more natural encoding in the following section.

### 3.3 Disjunctive Assertions

Our proof strategy for disjunctive assertions is similar to the \textit{near-Horn PROLOG} strategy developed by Loveland [14, 15]. Since this requires us to “restart” the proof procedure a number of times, the interpreter for OLD resolution in Figure 4 provides a good foundation for our work.

The basic strategy can be explained by reference to the proof in Figure 2, if we restrict our attention to the proof of ‘$\bot$’ from a database containing ‘$D_0$’. Suppose rule (12) is replaced by a rule with a Skolem function, and rule (13) is replaced by a rule with a single disjunct. We could then rewrite these rules as Horn clauses:

$$D_2(\text{sk}(v), v) \Leftarrow D_1(v)$$  \hfill (22)

$$\text{Mother}(u, v) \Leftarrow D_2(u, v)$$  \hfill (23)

Intuitively, we think of ‘\text{Mother}(u, v)’ as the \textit{prototypical disjunct} in rule (13). Let us now analyze the search for a Horn clause proof of ‘$\bot$’ from ‘$D_0$’ using (1)–(4), (11) and (22)–(23), and map this search onto the proof tree shown in Figure 2. The search would succeed using ‘$\text{Mother}(\text{sk}(\text{sk}(v)), \text{sk}(v))$’ and ‘$\text{Mother}(\text{sk}(v), v)$’, and the successful proof could be mapped onto the lower left branch of Figure 2 simply by reading proof steps (I4) and (I5) backwards. We call this the \textit{prototypical proof} of ‘$\bot$’ from ‘$D_0$’. For this proof to be correct for the original rules, however, we must also show that ‘$\bot$’ follows when ‘$\text{Mother}(\text{sk}(v), v)$’ is replaced by ‘$\text{Father}(\text{sk}(v), v)$’.
and when ‘Mother(sk(sk(v)), sk(v))’ is replaced by ‘Father(sk(sk(v)), sk(v))’. This disjunctive case analysis is the basic idea underlying our proof procedure.

prove2([Ans/Split<-[],DB,Suspend,Stack) :-
    (dsplit(DB,Split,Suspend,Stack);!,extend(DB,Suspend,Stack)).
prove2([Ans/Cases<-[],Frame|Pnode],DB,Suspend,Stack) :-
    copy_term(Frame,A/OldCases<-[Call|Goals]), Call=Ans,
    append(OldCases,Cases,NewCases),
    !,prove2([A/NewCases<-Goals|Pnode],DB,Suspend,Stack).
prove2(Node,DB,Suspend,Stack) :- Node = [<-[Goal|_]_],
    findall(Goal/[]<-[],db(Goal,DB),Fr1),
    findall(Goal/[]<-Body,rule(Goal,Body),Fr2),
    findall(Goal/Cases<-Body,drule(Goal,Cases,Body),Fr3),
    append(Fr2,Fr3,Fr),append(Fr1,Fr,Frames),
    suspend(Goal,Node,Suspend,NewSuspend),
    !,extend(DB,NewSuspend,[branch(Frames,Node)|Stack]).

dsplt(DB,[]),Suspend,Stack) :- !.
dsplt(DB,[A|Split],Suspend,Stack) :-
    suspensions(A,Suspend,NewSuspend,Branches),
    append(Branches,Stack,NewStack),
    !,extend([A|DB],NewSuspend,NewStack),
dsplt(DB,Split,Suspend,Stack).

Figure 6: An Interpreter for Disjunctive Assertions.

A simplified version of our interpreter for disjunctive assertions is shown in Figure 6. The three clauses of prove2 correspond to the three clauses of old in Figure 4, but with additional arguments. Assume, first, that DB is the empty list. Then the behavior of prove2 is similar to the behavior of old. The principal difference is due to the presence of the third findall in the third clause, which comes into play when Goal matches the prototypical disjunct in a disjunctive assertion — e.g., mother(U,V) in rule (13). A frame in this interpreter has the form Ans/Cases<-Goals, and the variable Cases is instantiated by drule to a list of the nonprototypical disjuncts in the disjunctive assertion — e.g., [father(U,V)] in rule (13). These cases are then appended together when the frame is processed by the second clause of prove2. (The code for extend is identical to the code in Figure 4, except for the extra arguments, and it is not shown here.)

When the interpreter reaches the first clause of prove2, it has found a prototypical proof and a list of nonprototypical cases. It is now the job of dsplit to add each of these cases to DB and restart the proof by calling extend with the current Stack as its argument. Subsequently, the first findall in the third clause of prove2 will call db(Goal,DB) to create a frame for any Goal that unifies with an atom in DB. (Note that these database frames are appended to the beginning of the list.) Eventually, for prove2 to succeed, it must find a prototypical proof that succeeds from DB without generating any nonprototypical cases, but prove2 fails when extend encounters an empty stack. Thus the first clause of prove2 calls extend again to search for a new prototypical proof whenever dsplit fails. This is the only backtracking point in the interpreter.

For this proof procedure to be correct, it is important not to discard any nodes that might subsequently succeed on DB, and this is the purpose of the predicates suspend and suspension. When the third clause of prove2 encounters a node that might later succeed on a nonprototypical
case, it adds a branch to the list `Suspend`, and `dsplit` checks this list to find any branches that should now be processed. It is possible to add several optimizations here, of course, by pruning search paths that would otherwise be redundant. We have incorporated these pruning heuristics into our actual interpreter, taking care not to remove any viable branches.

**Example 3.2:** To analyze the performance of `prove2`, let us consider a more natural encoding of the problem in Example 3.1. We will retain rules (16)--(18) for the definition of `SameColor`, but we will replace rules (14) and (15) by:

\[
\text{NotColorable} \leftarrow \text{Node}(x) \land \text{Node}(y) \land \text{Edge}(x, y) \land \text{SameColor}(x, y)
\]

and replace rules (19)--(21) by:

\[
\text{Node}(x) \Rightarrow \text{Red}(x) \lor \text{Green}(x) \lor \text{Blue}(x)
\]

Using cliques of size $n$ as a test case, and running Quintus PROLOG on a Sun 380, we have the following results for the query `NotColorable`:

<table>
<thead>
<tr>
<th>nodes</th>
<th>Constant DB</th>
<th>Variable DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.4 sec</td>
<td>1.6 sec</td>
</tr>
<tr>
<td>4</td>
<td>2.0 sec</td>
<td>2.3 sec</td>
</tr>
<tr>
<td>5</td>
<td>3.3 sec</td>
<td>3.8 sec</td>
</tr>
<tr>
<td>6</td>
<td>5.8 sec</td>
<td>8.2 sec</td>
</tr>
<tr>
<td>7</td>
<td>12.0 sec</td>
<td>15.2 sec</td>
</tr>
<tr>
<td>8</td>
<td>24.5 sec</td>
<td>27.2 sec</td>
</tr>
<tr>
<td>9</td>
<td>45.5 sec</td>
<td>54.7 sec</td>
</tr>
<tr>
<td>10</td>
<td>24.7 sec</td>
<td>27.1 sec</td>
</tr>
<tr>
<td>11</td>
<td>24.6 sec</td>
<td>27.0 sec</td>
</tr>
<tr>
<td>12</td>
<td>24.4 sec</td>
<td>27.1 sec</td>
</tr>
</tbody>
</table>

The double column for `time` in this table requires some explanation. The interpreter shown in Figure 6 does not work properly if there are atoms in `DB` containing free variables, as there would be in Example 2.3. It works fine if `DB` contains only constants, as it does in Example 3.2. We discuss this problem, and its solution, in the full version of the paper. For now, we simply note that the overhead for a version of `prove2` that handles variable `DB`s is negligible.

### 3.4 Embedded Implications and Disjunctive Assertions

To write an interpreter for the full language, we need to combine `prove1` and `prove2`. Our basic approach is to convert the SLD interpreter for embedded implications (shown in Figure 5) into an OLD interpreter, and then add this code to the interpreter for disjunctive assertions (shown in Figure 6). Thus, `prove3` has the same general structure as `prove2`, but it has two extra clauses to handle the case `forall(Ys, (Goal:-Hyp))`. One of these clauses corresponds to the third clause of `prove2` and the third and fourth clauses of `prove1`: It recognizes the occurrence of an implication as a goal, and it extends the current branch with a new auxiliary tableau and a new hypothetical database. The other clause corresponds to the first clause of `prove2`: It recognizes that the goal at the top of an auxiliary tableau has succeeded, and it attempts a disjunctive case analysis at this point. Several additional variables are now needed in the frame data structure: `HypDB` and `Vars` play roughly the same role as in `prove1`; the numerical level, $N$, in `prove1` is replaced by a list of
unique tableau identifiers, \([\text{tab}0, \text{tab}1, \ldots, \text{tab}N]\), so that the relationships between tableaux can be computed by a simple prefix matching algorithm \([5]\); and some additional bookkeeping information is required, as indicated below.

Although, in principle, a full disjunctive case analysis could be tried at the top of every auxiliary tableau, this would lead to redundant computation, since the alternative branches that might allow the case analysis to go through are already stored in \text{Stack} (or in \text{Suspend}) at the time the prototypical proof succeeds. We thus attempt a case analysis at the lowest possible point in the sequence of tableaux generated by the prototypical proof. On the other hand, it is sometimes necessary to move these proofs “up” the sequence of tableaux in order to unify a \(?x\) variable with a constant of type \(!y\). Our overall strategy, therefore, is to let the disjunctive case analysis “float” through a range of tableau levels, with the range initially determined by the prototypical proof and then progressively tightened as the disjunctive proof proceeds. We illustrate this strategy with the following artificial example:

**Example 3.3:** Let \(\mathcal{R}\) consist of the following rules:

\[
\begin{align*}
Q_0 & \leftarrow [G_1 \leftarrow D_1(x)] \\
G_1 & \leftarrow (\forall y)[G_2(y) \leftarrow D_2(y)] \\
G_1 & \leftarrow R_3(z) \\
G_1 & \leftarrow S_3(a) \\
G_2(v) & \leftarrow D_2(v) \land Q_3(z) \\
G_2(w) & \leftarrow R_4(w) \\
P_3(x, z) & \Rightarrow Q_3(z) \lor R_3(z) \lor S_3(x) \\
P_3(x, z) & \leftarrow D_1(x) \land Q_4(w) \\
P_4(w) & \Rightarrow Q_4(w) \lor R_4(w) \\
P_4(w) & \leftarrow
\end{align*}
\]

As a mnemonic device, we have used the same name for variables that are expected to unify with each other. We will see that the query ‘\(Q_0\)’ fails.

Figure 7 depicts the first prototypical proof computed by \text{prove3} in Example 3.3. (See [17] and [18] for similar illustrations of proofs involving embedded implications.) The proof begins with the goal ‘\(Q_0\)’ in the initial tableau \(T_0\). A unification with rule (26) generates the tableau \(T_1\) with a goal ‘\(G_1\)’ and a hypothetical database containing ‘\(D_1(\mathcal{\forall} x)\)’, and a unification with rule (27) generates the tableau \(T_2\) with a goal ‘\(G_2(\mathcal{\forall} y)\)’ and a hypothetical database containing ‘\(D_2(\mathcal{\forall} y)\)’. The proof in \(T_2\) now succeeds, using the prototypical disjunct ‘\(Q_3(z)\)’ from rule (32) and the prototypical disjunct ‘\(Q_4(w)\)’ from rule (34). Notice, however, that the proof of ‘\(P_4(w)\)’ would have succeeded at any level between \(T_0\) and \(T_2\), since it does not depend on the hypothetical databases, while the proof of ‘\(P_3(x, z)\)’ would have succeeded at any level between \(T_1\) and \(T_2\), since it depends on a formula in the hypothetical database of \(T_1\). The “free variables” associated with these disjunctive nodes are thus initially assigned a range of indices: The index of \(w\) has a lower bound \(T_0\) and an upper bound \(T_2\), and the index of \(z\) has a lower bound \(T_1\) and an upper bound \(T_2\). (Since the variable \(x\) unifies with the variable \(?x\) in the hypothetical database of \(T_1\), this index is fixed.)

The disjunctive case analysis for rule (32) now begins at the level of \(T_1\), adding ‘\(R_3(z)\)’ and then ‘\(S_3(x)\)’ to \(\text{DB}\). Although \text{Stack} (or \text{Suspend}) contains branches beginning in \(T_2\), e.g., those generated...
by rule (31), none of these would succeed here. Instead, `dsplit` succeeds in $T_1$ using rules (28) and (29), binding $?x$ to the constant ‘a’.

In the full version of this paper, we show that `prove1`, `prove2` and `prove3` are sound and (nondeterministically) complete when applied to the syntactically restricted language for which each is intended. Since `prove3` subsumes `prove1` and `prove2`, it could be used in all cases, even for the more restricted languages, although at some cost. Here are the comparisons for the query ‘$Q_1$’ given by (7)–(8) and the query ‘$Q_2$’ given by (10) in the “no grandparents” example:

<table>
<thead>
<tr>
<th>query</th>
<th>result</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><code>prove1</code></td>
</tr>
<tr>
<td>$Q_1$</td>
<td>fails</td>
<td>3.4 msec</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>succeeds</td>
<td>—</td>
</tr>
</tbody>
</table>

Although this is a very small example, it suggests (correctly) that the overhead from adding disjunctive assertions to embedded implications is greater than the overhead from adding embedded implications to disjunctive assertions.
4 Discussion

Our work is related to the work of several other researchers, some of which has already been discussed in the text. Our search heuristics combine some of the ideas of Beeson [1], Shankar [29] and Wallen [32], but the use of a syntactically restricted language permits a greater concentration of the heuristic effect of these ideas. Although Nadathur has written interpreters for embedded implications [24], and Loveland has written interpreters for disjunctive assertions [14, 15], the combination of these two interpreters appears to be novel. In the full version of this paper, we discuss this related work in greater detail.

There is a similarity between the proof method for finite disjunctions discussed in the present paper and the proof method for “infinite disjunctions” discussed in [18]. This similarity is exploited in [20], where the main example requires a combination of both techniques. We are currently extending these proof methods to handle the action language in [21] and the deontic language in [19], and we believe that our approach is applicable to a wide variety of commonsense reasoning tasks.

References


