

Implementation of a loop-free method for construction of counter-models for intuitionistic propositional logic

Roy Dyckhoff & Luis Pinto

Computer Science Division
University of St Andrews

{rd, luis}@dcs.st-and.ac.uk
<http://www-theory.dcs.st-and.ac.uk/~rd>

June 1996

Revised February 2001

Abstract

[PD] provided the theoretical background for a technique for constructing Kripke counter-models for intuitionistic propositional formulae, following [D]'s development of contraction-free sequent calculi for intuitionistic logic. We give here a Prolog implementation of this technique and illustrate its efficiency on some sample problems. The implementation can be uplifted electronically from the above web address.

1. Introduction

[D] introduced two cut-free calculi (**LJT** and **LJT***) for intuitionistic propositional logic with the property that there is a well-founded ordering (using the Dershowitz–Manna multi-set ordering theorem) on sequents such that for every primitive inference rule each premiss is smaller than the conclusion, thus ensuring that a backwards search for a proof of a sequent, from the root towards the leaves, always terminates.

The paper also introduced the terminology “contraction-free” for such calculi, because the contraction rule is no longer a primitive rule but is nevertheless, like the cut rule in these and other “cut-free” systems, admissible; and it covers other work on such calculi by Hudelmaier, Vorob’ev, Lincoln, Shankar and Scedrov.

[PD] showed that the multi-succedent calculus **LJT*** is a satisfactory basis for a non-looping method to construct Kripke trees as counter-models for non-provable formulas of intuitionistic propositional logic. It gave a calculus CRIP (Calculus for Refutation of Intuitionistic Propositions) in which “refutations” are inductively defined; from these refutations Kripke counter-models can be extracted, rather as one extracts lambda terms from natural deduction proofs.

In the present report our purpose is to present a Prolog implementation, tested under SICSTUS Prolog 2.1 and easily portable to other Prologs, of this method. We also give some background on Kripke models, some experimental results and some illustrations on the efficiency of the implementation.

2. Background on Kripke models

Kripke structures (in particular, Kripke trees) are widely used as models or counter-models for non-classical logics. The basic theory is as follows, for the usual formalisation of intuitionistic propositional logic with $\&$, $\#$ and \rightarrow as primitive binary connectives and \perp as a nullary connective representing absurdity. Negation \sim and bi-implication \leftrightarrow are defined in the usual way.

Let K and L be sets, and let \leq be a binary relation on K . [We make no further assumptions yet about \leq .] A *monotone relation* from K to L is given by any of the following equivalent situations:

- a *monotone* function $f : K \rightarrow \wp(L)$, i.e. a function from K whose values are subsets of L , s.t. for all $k, k' \in K$, $k \leq k'$ implies $f(k) \subseteq f(k')$.
- a binary relation \gg (“forces”) between K and L , s.t.
for all $k, k' \in K, l \in L$ ($k \gg l$ and $k \leq k'$) implies $k' \gg l$
- a function $\phi : L \rightarrow U(K)$, where $U(K)$ consists of the *upwards-closed* subsets of K , i.e. subsets H of K s.t. $k \in H$ and $k \leq k'$ implies $k' \in H$.

The correspondence between the three situations is given by

$$l \in f(k) \text{ iff } k \gg l \text{ iff } k \in \phi(l)$$

We shall switch between the three views of the relation as convenient.

Now let L be a set whose elements we call “proposition variables”. A *Kripke interpretation* of L consists of a set K , a binary relation \leq on K and a monotone relation \gg from K to L . Let L^* be the language generated from L by the nullary constant \perp and the binary connectives $\&$, $\#$ and \rightarrow . The interpretation is *transitive* iff \leq is a transitive relation. The elements of K are variously called *worlds* or *states of knowledge*. A world k' with $k \leq k'$ is said to be *accessible* from k .

Lemma. If (K, \leq, \gg) is a transitive interpretation of L , then there is a unique Kripke interpretation (K, \leq, \gg^*) of L^* satisfying the conditions (for all $k \in K, l \in L$ and $A, B \in L^*$)

- (i) $k \gg^* l$ iff $k \gg l$
- (ii) $k \gg^* (A\&B)$ iff $k \gg^* A$ and $k \gg^* B$
- (iii) $k \gg^* (A\#B)$ iff $k \gg^* A$ or $k \gg^* B$
- (iv) $k \gg^* (A \rightarrow B)$ iff for all $k' \geq k$, $k' \gg^* A$ implies $k' \gg^* B$
- (v) $\text{not}(k \gg^* \perp)$.

Proof. Uniqueness: by induction on the structure of the formulae.

Existence: the first four conditions suffice to define the relation \gg^* . We must check that \gg^* is a monotone relation from K to L^* , i.e. that if $k \gg^* A$ and $k \leq k'$ then $k' \gg^* A$; this needs induction on A and, for condition (iv), the transitivity of \leq . **QED.**

Lemma. If (K, \leq, \gg) is a transitive interpretation of L , if $k \leq k'$ and if $k \gg^* A$, then $k' \gg^* A$.

Proof. By induction on the structure of A . **QED.**

A *Kripke tree* (for L) is a Kripke interpretation (of L) such that the ordered set (K, \leq) is a tree, i.e. \leq is a reflexive transitive binary relation on K with a least element k_0 (its *root*) so that $K \setminus k_0$ is a disjoint union of Kripke trees. A Kripke interpretation of L is a *model* of a formula A iff for every k in K, $k \gg^* A$; and is a *counter-model* of A otherwise, i.e. iff for some k in K $\text{not}(k \gg^* A)$, i.e. iff $\text{not}(k_0 \gg^* A)$.

3. Implementation

3.0 Introduction

The technique described in [DP] is implemented in SICSTUS prolog 2.1 and tested on a SUN SPARCStation 10. The main code is divided into 6 separate files `syntax.pl`, `library.pl`, `kripke.pl`, `print.pl`, `crip.pl` and `crip2.pl`, conveniently loaded by invoking the interpreter with the goal `[load]`. The file `sample.pl` of sample problems can be used by using the goal `[sample]`. Timings given later use compiled code. The timings are generated by `timings.pl` (loaded also by `[load]`) and by `[tests]`.

3.1 Syntax

The syntax is organised using the usual Prolog technique of operator directives, making for example the `&` and `#` symbols right associative and `-->` left associative with lower precedence (i.e. higher Prolog precedence), so `p-->q&r` abbreviates `(p-->(q&r))`.

```
% syntax.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SYNTAX

:-op(500,xfy,[&,#]).
:-op(600,yfx,-->).
:-op(650,yfx,<->).
:-op(700,yfx,=>).
:-op(400,fy,~).

% absurdity is represented by the atom 'ff'.
% atoms are represented by Prolog atoms.
```

3.2 Library Utilities

```
% library.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LIBRARY utilities

member(H,[H|_]).
member(H,[_|T]):-
    member(H,T).

not_member(_,[]).
not_member(X,[H|T]):-
    X\==H,
    not_member(X,T).

member3(H,[H|T],T).
member3(H,[H1|T],[H1|T1]):-
    member3(H,T,T1).
```

```

append( [],L,L).
append( [H|T],L,[H|T1]):-
    append(T,L,T1).

disjoint([],_).
disjoint([H|T],L):-
    not_member(H,L),
    disjoint(T,L).

subset( [],_ ).
subset( [Elem|Set],Set1 ):-
    member(Elem,Set1),
    subset(Set,Set1).

add_set( [],Set,Set ).
add_set( [Elem|Elems],Set,Set1 ):-
    ( member(Elem,Set),!,
      Set2=Set
    ;
      Set2=[Elem|Set]
    ),
    add_set(Elems,Set2,Set1).

```

3.3 Kripke Semantics

```

% kripke.pl
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEPENDENCIES
% Uses definition of syntax in
%     syntax.pl

% Uses predicates defined in
%     library.pl:
%     member/2, subset/2.

% The internal representation for a Kripke structure is: k(Ats,Ks)
% where
% Ats is the list of atoms forced at the world
% Ks is the list of Kripke structures accessible from the world.
% e.g. (adding numeric labels for readability):
%
%       1 |=p,q   2 |= p,r
%       / \
%      /   \
%     /     \
%    /       \
%   /         \
%  /           \
% /             \
%0
%
% is represented by:
%       k([], [k([p,q],[ ]),k([p,r],[ ])])
%
%=====
% check(KS, F)
% -----
% Check that a counter-model KS of formula F really is one
% i.e. that it really is a Kripke structure
% and that its root world does NOT force F
% -----
check( KS, F ) :-
    ( ks(KS),
      !,
      nl, write('THE STRUCTURE IS A KRIPKE STRUCTURE.')
    ;
      nl, write('NOT A KRIPKE STRUCTURE. AAARRRRGGGHHH!!!'), fail
    ),

```

```

( forces(F,KS),
  !,
  nl,write('but NOT A COUNTER_MODEL. AAARRRRGGGHHH!!!'), fail
;
  nl,write('IN FACT A COUNTER-MODEL.')
).

%=====

% ks( K )
% -----
% Checks that K is a Kripke structure
% i.e. the atoms true at each node are a subset of
% those true at any higher node
% -----

ks( K ) :- ks1( K, []).

ks1( k(Ats, Ks), Ats1 ) :-
  subset(Ats1, Ats),
  ks2(Ks, Ats).

ks2( [], _ ).
ks2( [ K | Ks ], Ats ) :-
  ks1(K, Ats),
  ks2(Ks, Ats).

%=====

% forces(A,K)
% -----
% K forces A
% -----
forces( A & B, K ):-
  forces(A,K), forces(B,K).

forces( A # B, K ):-
  ( forces(A,K),! ; forces(B,K) ).

forces( A --> B, K ):-
  force_impl(A,B,K).

forces( ff, _ ) :-
  !, fail.

forces( At, k(Ats, _ ) ):-
  atomic(At),
  member(At,Ats),
  !.

%=====

% force_impl(A,B,K)
% -----
% checks that for every world accessible from K,
% if A is forced then so is B
% -----
force_impl( A, B, K ) :-
  forces(A, K),
  !,
  forces(B, K).

force_impl( A, B, k(_, Ks) ) :-

```

```

force_impl_all(Ks, A, B).

force_impl_all( [], _, _ ).
force_impl_all( [ K | Ks ], A, B ) :-
    force_impl(A,B,K),
    force_impl_all(Ks, A, B).

```

3.4 CRIP (Calculus for Refutation of Intuitionistic Propositions)

We modify slightly the calculus given in [PD]. That gave separately both an *axiom* rule and a rule (11), of which the *axiom* rule is a special case (with $m=n=0$). The implementation below combines the two together. It also corrects the omission in [PD] of the proviso for rule (11) that absurdity ff should not be in the antecedent. (This is implicit in the proviso, since ff is an atom: but it should have been made clear.)

```

% cripl.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculus for Refutation of Intuitionistic Propositions
% described in paper by Pinto and Dyckhoff
% Symposia Gaussiana conference at Munich 1993
% available from http://www-theory.dcs.st-and.ac.uk/~rd
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Idea is either to refute an intuitionistic formula with an exhibited
% counter-model
% or to report "provable"
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% DEPENDENCIES
%
% Assumes that the syntax has been defined by
%     syntax.pl

% Uses the following predicates defined in
%     library.pl
%     member/2; subset/2; add_set/3; member3/3
%     append/3; not_member/2; disjoint/2

% Uses the following predicates defined in
%     kripke.pl
%     check/2.

% Uses the following predicates defined in
%     cripl2.pl
%     atom_impl_or_atoms/3; just_atoms/1; unfold_neg/2;
%     coll_nested_impl/3; coll_impl_right/3; coll_atoms/2.

% Uses the following predicates defined in
%     print.pl
%     pretty_print/2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TOP LEVEL -----
%
% F must be a formula
% if it is provable, then 'Provable' is output
% otherwise a counter-model KS is output
%-----
decide( F ):-
    unfold_neg(F,G),
    nl, write(F),

```

```

nl,
( unprovable([] => [G],KS),
  write('Counter Model: '),
  pretty_print(KS),
  check(KS, G)
;
  write('Provable')
), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the CRIP calculus -----

unprovable( Ant => Succ,KS ):-
  member3(A & B,Ant,Ant1),
  !,

%           A,B,T =/=> T'
%           ===== Rule (1)
%           A&B,T =/=> T'

  add_set([A,B],Ant1,Ant2),
  unprovable(Ant2 => Succ,KS).

%-----

unprovable( Ant => Succ,KS ):-
  member3(A # B,Succ,Succ1),
  !,

%           T =/=> T',A,B
%           ===== Rule (6)
%           T =/=> T',A#B

  add_set([A,B],Succ1,Succ2),
  unprovable(Ant => Succ2,KS).

%-----

unprovable( Ant => Succ,KS ):-
  member3(A --> B,Ant,Ant1),
  ( atomic(A),
    member(A,Ant1),
    !,

%           p,B,T =/=> T'
%           ===== [p atomic] Rule (7)
%           p,p-->B,T =/=> T'

  add_set([B],Ant1,Ant2),
  unprovable(Ant2 => Succ,KS)
;
  (
    A = C&D,
    !,

%           C-->(D-->B),T =/=> T'
%           ===== Rule (9)
%           (C&D)-->B,T =/=> T'

  add_set([C-->(D-->B)],Ant1,Ant2),
  unprovable(Ant2 => Succ,KS)
;
  A = C#D,

```

```

!,
%
%           C-->B,D-->B,T =/=> T'
%           ===== Rule (8)
%           (C#D)-->B,T =/=> T'
%
%           add_set([C-->B,D-->B],Ant1,Ant2),
%           unprovable(Ant2 => Succ,KS)
%       )
%   ).
%-----
unprovable( Ant => Succ,KS ):-
    member3(A & B,Succ,Succ1),
    !,
    (
%           T =/=> T',A
%           ===== Rule (2)
%           T =/=> T',A&B
%
%           add_set([A],Succ1,Succ2),
%           unprovable(Ant => Succ2,KS),!
%       ;
%           T =/=> T',B
%           ===== Rule (3)
%           T =/=> T',A&B
%
%           add_set([B],Succ1,Succ2),
%           unprovable(Ant => Succ2,KS)
%   ).
%-----
unprovable( Ant => Succ,KS ):-
    member3(A # B,Ant,Ant1),
    !,
    (
%           A,T =/=> T'
%           ===== Rule (4)
%           A#B,T =/=> T'
%
%           add_set([A],Ant1,Ant2),
%           unprovable(Ant2 => Succ,KS),!
%       ;
%           B,T =/=> T'
%           ===== Rule (5)
%           A#B,T =/=> T'
%
%           add_set([B],Ant1,Ant2),
%           unprovable(Ant2 => Succ,KS)
%   ).
%-----
unprovable( Ant => [A-->B],KS ):-
    !,
%
%           T,A =/=> B
%           ===== Rule (11') [Extra optimisation..]
%           T =/=> A-->B
%
%           add_set([A],Ant,Ant1),

```

```

unprovable(Ant1 => [B],KS).

%-----
unprovable( Ant => Succ,KS ):-
%
%           B,T =/=> T'
%           ----- Rule (10)
%           (C-->D)-->B,T =/=> T'
%
member3(( _-->_ )-->B,Ant,Ant1),
add_set([B],Ant1,Ant2),
unprovable(Ant2 => Succ,KS).

%-----

unprovable( Ant => Succ, k(As,KSS) ):-

%   D1-->B1,T1 =/=> C1-->D1
%   ....
%   Dn-->Bn,Tn =/=> Cn-->Dn
%   T',E1 =/=> F1
%   ....
%   T',Em =/=> Fm
%   -----
%   (C1-->D1)-->B1,...,(Cn-->Dn)-->Bn,T =/=> E1-->F1,...,Em-->Fm,T''

% where T' is the antecedent Ant of conclusion
%   Ti is T' with (Ci-->Di)-->Bi removed

% Proviso:
% Every formula in T is either atomic or an atomic implication.
% None of the antecedents of the atomic implications are equal to
%   any of the atomic formulae
% T' contains only atomic formulae.
% T does NOT contain ff
% T and T'' are disjoint.

coll_nested_impl(Ant,NIs,Ant1), % so Ant1 is T
atom_impl_or_atoms(Ant1, AIs, As),
% check that T is just atoms or atomic impls
disjoint(AIs, As),
coll_impl_right(Succ,IsR, Succ1),
just_atoms(Succ1),
not_member(ff, Ant1),
disjoint(As, Succ1),
unprovable_list1(NIs,Ant, KSS1),
unprovable_list2(IsR,Ant, KSS2),
append(KSS1,KSS2,KSS).

%=====

% unprovable_list1(NIs, Ant, Models)
%-----
% For each nested implication ((C-->D)-->B) in NIs
% constructs counter-model for D-->B, Ant* => C-->D
% [ in fact for C, D-->B, Ant => D ]
% where Ant* is Ant with the nested implication removed
% Returns list of such models
%-----
unprovable_list1( [],_,[] ).
unprovable_list1( [Imp|NIs],Ant,[KS|KSS] ):-
Imp = (C -->D) --> B,
member3( Imp, Ant, Antstar),

```

```

unprovable( [C, D-->B | Antstar] => [D], KS),
unprovable_list1(NIs,Ant,KSs).

%=====

% unprovable_list2(IsR, Ant, Models)
%-----
% For each implication E-->F in IsR
% constructs counter-model for E, Ant => F
% returns list of such models
%-----
unprovable_list2( [],_,[] ).
unprovable_list2( [E-->F|IsR],Ant,[KS|KSs] ):-
    add_set([E],Ant,Ant1),
    unprovable(Ant1 => [F],KS),
    unprovable_list2(IsR,Ant,KSs).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% crip2.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% various utilities to support crip.pl

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEPENDENCIES
%
% Assumes that the syntax has been defined by
%     syntax.pl

% This file uses the following predicates defined in
%     library.pl
%     member/2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% atom_impl_or_atoms(First, Second, Third)
% -----
% Checks that First consists of
%     either atomic implications (P --> B) or atoms Q
% collects all the atoms P into Second and atoms Q into Third
% -----
atom_impl_or_atoms( [],[], []).
atom_impl_or_atoms( [(P --> _)|Fs],[P|Ats1], Ats2 ):-
    atomic(P),
    !,
    atom_impl_or_atoms(Fs, Ats1, Ats2).
atom_impl_or_atoms( [Q|Fs], Ats1, [Q|Ats2] ):-
    atomic(Q),
    atom_impl_or_atoms(Fs, Ats1, Ats2).

% =====

% just_atoms( First)
%-----
% Checks that First consists of just atoms)
% -----
just_atoms( [] ).
just_atoms( [F|Fs] ):-
    atomic(F),
    just_atoms(Fs).

% =====

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%Expanding negations and equivalences

unfold_neg( A & B,A1 & B1 ):-
  unfold_neg(A,A1),
  unfold_neg(B,B1).
unfold_neg( A # B,A1 # B1 ):-
  unfold_neg(A,A1),
  unfold_neg(B,B1).
unfold_neg( A --> B,A1 --> B1 ):-
  unfold_neg(A,A1),
  unfold_neg(B,B1).
unfold_neg( A <-> B,(A1 --> B1)&(B1 -->A1) ):-
  unfold_neg(A,A1),
  unfold_neg(B,B1).
unfold_neg( ~ A,A1 --> ff ):-
  unfold_neg(A,A1).
unfold_neg( A,A ):-
  atomic(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Collecting implications or atoms
% -----
% coll_nested_impl(First, Second, Third)
% -----
% This splits First into Second and Third
% Second will consist of all the nested implications
% Third is the remainder
% -----
coll_nested_impl( [NImp|L],[NImp|L1],L2 ):-
  NImp = (( _ --> _ ) --> _),
  !,
  coll_nested_impl(L,L1,L2).
coll_nested_impl( [F|L],L1,[F|L2] ):-
  coll_nested_impl(L,L1,L2).
coll_nested_impl( [],[],[] ).
% -----
% coll_impl_right(First, Second, Third)
% -----
% This splits First into Second and Third
% Second will consist of all the implications
% Third is the remainder
% -----
coll_impl_right( [F|L],[F|L1], L2 ):-
  F = ( _ --> _ ),
  !,
  coll_impl_right(L,L1, L2).
coll_impl_right( [F|L],L1,[F|L2] ):-
  coll_impl_right(L,L1,L2).
coll_impl_right( [],[],[] ).
% -----
% coll_atoms(First, Second)
% -----
% This collects all the atoms in First
% and puts them into Second
% -----
coll_atoms( [A|L],[A|L1] ):-
  atomic(A),
  !,
  coll_atoms(L,L1).
coll_atoms( [_|L],L1 ):-
  coll_atoms(L,L1).
coll_atoms( [],[] ).

```

3.5 Pretty printing of Kripke trees

The pretty printing of Kripke trees is straightforward: see later for an illustration of how the trees appear. Trees as constructed have no labels indicating the nodes: increasing numeric labels are output by the printing routine. Here is the implementation:

```
% print.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%PRETTY-PRINTER

pretty_print( KS ) :-
    pretty_print(KS, 0, 0,_).

% N is the indentation
% W is the world index
% final W2 is the next world index to use
pretty_print( k(Ats,Ks) ,N, W, W2 ):-
    print_world(Ats,N, W, W1),
    N1 is N+1,
    pretty_print_list(Ks,N1, W1, W2).

pretty_print_list( [],_, W, W ).
pretty_print_list( [H|T],N, W, W2 ):-
    pretty_print( H, N, W, W1),
    pretty_print_list( T,N, W1, W2).

print_world( Ats,N, W, W1 ):-
    nl,
    printN(N),
    write(W),
    write(' |= '),
    W1 is W + 1,
    write(Ats).

printN( 0 ):-!.
printN( N ):-
    write(' '),
    N1 is N-1,
    printN(N1).
```

3.6 Timings

```
%timings.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%
% decides a formula F 100 times and outputs it,
% the answer and the total time in msec.

results( F ) :-
    times(Times),
    statistics(runtime, _),
    unfold_neg(F, G),          % No point doing THIS 100 times
    doit(G, Ans, Times),
    statistics(runtime, [_,Time]),
    write(F), nl,
    write(Ans), nl,
    write(Time), nl.

times(10).

doit(_, _, 0).
doit(F, Ans, Times) :-
    provable_or_not(F, Ans),
    Times1 is Times - 1,
    doit(F, Ans, Times1).

provable_or_not( F, Ans) :-
    unprovable( [] => [F], _ ),
    !,
    Ans = unprovable.

provable_or_not( _, Ans) :-
    Ans = provable.
```

3.6 Sample input and output

This is a combination of part of the input from the file `sample.pl` and the corresponding output, edited by removal of information other than some counter-models.

```
% sample.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- decide( (~ ~p-->p) --> (p # ~p) --> (~p # ~ ~p) --> (~ ~p # (~ ~p-->p)) ).
% should be non-provable
Response:
~ ~p-->p-->p# ~p--> ~p# ~ ~p--> ~ ~p#(~ ~p-->p)
Counter Model:
0 |= []
  1 |= []
    2 |= [p]
      3 |= [p]
        4 |= []
          5 |= [p]
            6 |= [p]
              7 |= []
                8 |= [p]
                  9 |= []

:- decide( p # (p --> q) # (q --> r) # (r --> s) # ~s ).
% should be non-provable
Response:
p#(p-->q)#(q-->r)#(r-->s)# ~s
Counter Model:
0 |= []
  1 |= [s]
  2 |= [r]
  3 |= [q]
  4 |= [p]

:- decide( (p-->q)#(q-->p) ).
% should be non-provable
Response:
(p-->q)#(q-->p)
Counter Model:
0 |= []
  1 |= [q]
  2 |= [p]

:- decide( p<->q<->(q# ~p)& ~q#p ).
% should be non-provable
Response:
p<->q<->(q# ~p)& ~q#p
Counter Model:
0 |= []
  1 |= [q,p]

:- decide( p<->q<->r<->(p<->(q<->r)) ).
% should be non-provable
Response:
p<->q<->r<->(p<->(q<->r))
Counter Model:
0 |= []
  1 |= [p]

:- decide( ~ ~(((p --> q) --> p) --> p) ).
% should be provable (Peirce's law, after DN)
[Response edited out.]
```

4. Results

There are three issues here: the speed of the search, the space requirements of the search and the size of the counter-model so constructed. We have made attempts neither to measure the space requirements nor to minimise the size of the counter-model. Nor have we made much effort to improve the Prolog code by extensive use of first-argument indexing. The speed of the search could be compared with the speed of an implementation of a theorem-prover based on one of the calculi in [D] or any of the many other calculi; this endless task is left for another occasion.

We measure therefore for various formulae F the time in milliseconds for the achievement of the goal `unprovable([] => F, _)` or its failure, 100 times. Timings are on a SUN SparcStation 10; they are runtimes, i.e. [SP] "CPU time used while executing, excluding time spent garbage collecting, stack shifting or in system calls". Timings ignore the preprocessing time, i.e. expansion of negations and bi-implications.

The results are as follows:

```
p-->q-->p-->p
unprovable
79
```

```
(p-->q)#(q-->p)
unprovable
70
```

```
p-->q<-> ~p#q
unprovable
89
```

```
(~ ~p-->p)# ~p# ~ ~p
unprovable
170
```

```
(~ ~p-->p)# ~p# ~ ~p#(~ ~q-->q)# ~q# ~ ~q
unprovable
319
```

```
~ ~p-->p-->p# ~p--> ~p# ~ ~p--> ~ ~p#(~ ~p-->p)
unprovable
5449
```

```
(p-->q)#(q-->r)#(r-->s)#(s-->p)
unprovable
120
```

```
p-->q-->q-->(r-->q-->q)-->(p-->r-->q-->q)
unprovable
699
```

```
p#(p-->q)#(q-->r)#(r-->s)# ~s
unprovable
169
```

```
p&(q-->r)-->s<->(~p#q#s)& ~p# ~r#s
unprovable
160
```

```
p<->q<->(q# ~p)& ~q#p
unprovable
```

160

$p \leftrightarrow q \leftrightarrow r \leftrightarrow (p \leftrightarrow (q \leftrightarrow r))$
unprovable
1200

$\sim \sim (\sim a \# \sim b) \leftrightarrow \sim a \# \sim b \leftrightarrow \sim \sim (\sim a \# \sim b) \# \sim (\sim a \# \sim b) \leftrightarrow \sim \sim (\sim a \# \sim b) \# \sim (\sim a \# \sim b)$
unprovable
5420

$\sim \sim (p \leftrightarrow q \leftrightarrow p \leftrightarrow p)$
provable
199

$\sim \sim (p \leftrightarrow q) \leftrightarrow (\sim \sim p \leftrightarrow \sim \sim q)$
provable
409

$(q \leftrightarrow r) \& (r \leftrightarrow p \& q) \& (p \leftrightarrow q \# r) \leftrightarrow (p \leftrightarrow q)$
provable
440

$p \# q \& r \leftrightarrow (p \# q) \& p \# r$
provable
580

$\sim \sim (p \# \sim p)$
provable
140

$p \leftrightarrow (q \leftrightarrow r) \leftrightarrow (p \leftrightarrow q \leftrightarrow (p \leftrightarrow r))$
provable
109

Of these, the two hardest each take about 5 seconds for 100 executions, i.e. 0.05 seconds each. A reading of 160, for example, means about 1.6 msec per execution. However, the following problem (suggested by Prof. N.J. de Bruijn) is found much harder (about 1.21 seconds per execution):

$((a \leftrightarrow b) \leftrightarrow a \& b \& c) \& ((b \leftrightarrow c) \leftrightarrow a \& b \& c) \& ((c \leftrightarrow a) \leftrightarrow a \& b \& c) \leftrightarrow a \& b \& c$
provable
121000

5. Conclusion

We have documented an implementation of the loop-avoidance method justified theoretically in [D] and [PD]. There has been no attempt to enhance the performance with additional techniques to speed up the search or to generate (as in [S]) minimal models: there is potentially much further interesting work to be done here. We do not regard the implementation as a proof search method; it does not return proofs but (Kripke trees as) counter-models.

Whether the performance is satisfactory depends on the purpose for which the implementation is intended. Our initial interest in this topic arose from a wish to understand better the relationship between backtracking and model generation and a separate wish to have an implementation for incorporation in a teaching tool, where the construction of models may help students understand why certain formulae are not provable. For these purposes the implementation is satisfactory except for the unnecessarily large counter-models, a topic addressed elsewhere [S].

References (see also [D] and [PD])

- [D] Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic, *Journal of Symbolic Logic* 57 (1992), 795–807.
- [PD] Pinto, L. & Dyckhoff, R.: Loop-free construction of counter-models for intuitionistic propositional logic, *Symposia Gaussiana, Conf. A*, Eds.: Behara/Fritsch/Lintz, Walter de Gruyter & Co, Berlin, New York 1995, 225–232.
- [S] Stoughton, A.: “porgi”: a Proof-Or-Refutation Generator for Intuitionistic propositional logic, *CADE 13 Workshop on Proof Search in Type theoretic languages*, Rutgers University (New York, USA), (1996), to appear.
- [SP] SICStus Prolog User’s Manual, release 3 # 0, SICS, Kista, Sweden, June 1995.