Formula Transformers and Combinatorial Test Generators for Propositional Intuitionistic Theorem Provers

Paul Tarau

University of North Texas

CLA'2019

Research supported by NSF grant **1423324**

Overview

We describe algorithms supporting a combinatorial testing framework for intuitionistic propositional logic (IPC).

- formula generators
  - known to be provable formulas: types inferred for lambda terms
  - formulas for full IPC and several sub-languages
  - all-term generators
  - random term generators
- formula transformers
  - to make prover computations easier
  - to make tests harder $\Rightarrow$ better at catching bugs
- lightweight theorem provers for IPC and several sub-languages
- Prolog implementation available at:
  https://github.com/ptarau/TypesAndProofs.
  Python implementation is available at:
  https://github.com/ptarau/PythonProvers.

The combinatorial testing framework

Combinatorial testing, automated

- testing correctness:
  - a false positive: it is not a tautology, but the prover proves it
  - a false negative: it is a tautology but the prover fails on it
  - no false positive: a prover is sound
  - no false negative: a prover is complete
  - soundness and completeness relative to a "gold standard"
- helpers:
  - intuitionistic tautologies are also classical, so if it is not classical it cannot be intuitionistic
  - crossing the Curry-Howard bridge: types of lambda terms or combinator expressions are tautologies, when seen as formulas
- all-term vs. random testing
  - all typed terms of a given size, known to be tautologies
  - all formulas up to given size: a mix of non-tautologies and tautologies (tautologies are fewer and fewer with size)
  - random simply typed lambda terms and combinator expressions
  - random IPC formulas

Components of the testing framework

- finding false negatives by generating the set of simply typed normal forms of a given size
- finding false positives by generating all implicational formulas/type expressions of a given size
- testing against a trusted reference implementation
- random simply-typed terms with Boltzmann samplers
- random simply typed combinator expressions with Rémy's algorithm
- generating random implicational formulas
- performance/scalability tests

Generating tautologies via the Curry-Howard correspondence

The Curry-Howard isomorphism

it connects:

- the implicational fragment of propositional intuitionistic logic
- types in the *simply typed lambda terms, in normal form*
- types of *simply typed combinator expressions*

complexity of “crossing the bridge”, different in the two directions

- a (low polynomial) type inference algorithm associates a type (when it exists) to a lambda term
- PSPACE-complete algorithms associate lambda terms as inhabitants to a given type expression

$\Rightarrow$

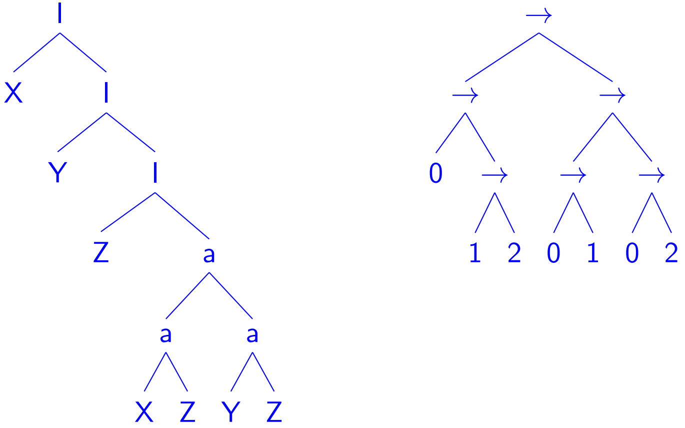
- a lambda term (typically in normal form) can serve as a witness for the existence of a proof for the corresponding tautology in the implicational fragment of propositional intuitionistic logic

Prolog notations in upcoming code snippets

- Prolog programming background:
  - variables will be denoted with uppercase letters
  - the pure Horn clause subset +
  - well-known built-in predicates like `memberchk/2` and `select/3`, `call/N`), CUT (!) and `if-then-else` constructs
  - Definite Clause Grammars (DCGs): a macro expansion mechanism, e.g., from `a-->b,c,d` to `a(S0,S3):-b(S0,S1),c(S1,S2),d(S2,S3)`
- lambda terms: **a/2**=application, **l/2**=lambda binders with a variable as its first argument, an expression as second and *logic variables* representing the leaf variables bound by a lambda
- type expressions (also seen as implicational formulas): binary trees with the function symbol "`->/2`", atoms or integers as their leaves
- full IPC formulas using the operators: `~`, `->`, `&`, `v`, `<->`

An example of a term and its type , represented as trees

the **S** combinator (left) and its type (right, with integers as leaves):



A generator of simply-typed normal forms and their types

```
typed_nf(N,X:T):-typed_nf(X,T,[],N,0).

pred(SX,X):-succ(X,SX).

typed_nf(l(X,E),(P->Q),Ps)-->pred,typed_nf(E,Q,[X:P|Ps]).
typed_nf(X,P,Ps)-->typed_nf_no_left_lambda(X,P,Ps).

typed_nf_no_left_lambda(X,P,[Y:Q|Ps])--> agrees(X:P,[Y:Q|Ps]).
typed_nf_no_left_lambda(a(A,B),Q,Ps)-->pred,pred,
  typed_nf_no_left_lambda(A,(P->Q),Ps),
  typed_nf(B,P,Ps).

agrees(P,Ps,N,N):-member(Q,Ps),unify_with_occurs_check(P,Q).
```

Trimming out the lambda terms: a generator of tautologies

```
impl_taut(N,T):-impl_taut(T,[],N,0).

impl_taut((P->Q),Ps)-->pred,impl_taut(Q,[P|Ps]).
impl_taut(P,Ps)-->impl_taut_no_left_lambda(P,Ps).

impl_taut_no_left_lambda(P,[Q|Ps])--> agrees(P,[Q|Ps]).
impl_taut_no_left_lambda(Q,Ps)-->pred,pred,
  impl_taut_no_left_lambda((P->Q),Ps),
  impl_taut(P,Ps).

?- countGen2(impl_taut,15,Rs).
Rs=[1,2,3,7,17,43,129,389,1245,4274,14991,55289,210743,826136,3354509]
```

- not the same as: OEIS A224345 using “natural size” of λ-terms
- we use here size 0 for variables, 1 for lambdas, 2 for applications

Examples of implicational tautologies

after "numbering variables" as natural numbers:

```
implTaut(N,T):-impl_taut(N,T),natvars(T).

?- implTaut(4,T).
T =  (0->1->2->3->3) ;
T =  (0->1->2->3->2) ;
T =  (0->1->2->3->1) ;
T =  (0->1->2->3->0) ;
T =  (0->(0->1)->1) ;
T =  ((0->1)->0->1) ;
T =  (((0->0)->1)->1) .
```

Challenges to generating hard tautologies

- asymptotic sparseness of the intuitionistic tautologies in the set of all formulas
- why: they are a subset of classical tautologies, which are already sparse
- asymptotic sparseness of typable lambda terms and combinator expressions
- automatically generated formulas are often too easy for the best provers

The formula transformers

Defining transformers to "equiprovable" formulas

The transformers: Why?

- we have known-to-be tautologies, but they are too easy for the provers
- $\rightarrow$ let's make them harder!
- we have formulas that we do not know as provable or unprovable
- $\rightarrow$ we want to perform simplifications to facilitate the work of the provers
- $\rightarrow$ converting between equivalent representations w.r.t. provability

Correctness of the transformers:

- agreement on the success of a correct prover before and after a transformation is applied

Implicational formulas as nested Horn Clauses

- equivalence between:
  - $B_1 \to B_2 \to \ldots \to B_n \to H$ and
  - H <- $[B_1, B_2, \ldots, B_n]$, with the list seen as conjunction
- H is the *atomic* formula ending a chain of implications
- we can recursively transform an implicational formula:

```
toAHorn((A->B),(H<-Bs)):-!,toAHorns((A->B),Bs,H).
toAHorn(H,H).

toAHorns((A->B),[HA|Bs],H):-!,toAHorn(A,HA),toAHorns(B,Bs,H).
toAHorns(H,[],H).

?- toAHorn(((0->1->2)->(0->1)->0->2),R).
R =  (2<-[(2<-[0, 1]),  (1<-[0]), 0]).

?- toAHorn(((0->1->2->3->4)->(0->1->2)->0->2->3),R).
R =  (3<-[(4<-[0, 1, 2, 3]),  (2<-[0, 1]), 0, 2]).
```

- also, note that the transformation is reversible!

A nested Horn Clause transformer for more general formulas

- the disjunction-free fragment of IPC i.e. the propositional *N-Prolog* subset can be converted to a list of nested Horn clauses (with `~p` expanded to `p->false`)
- → solvable by our nested Horn Clause provers

```
?- toNestedHorn((a<->b& ~c)->(d&e<->f),R).
R = [(f:-[d,e,(a:-[b,(false:-[c])]),(b:-[a]),(false:-[a,c])]),
     (d:-[f,(a:-[b,(false:-[c])]),(b:-[a]),(false:-[a,c])]),
     (e:-[f,(a:-[b,(false:-[c])]),(b:-[a]),(false:-[a,c])])]
```

- this is, in fact, the propositional subset of Dov Gabbay's N-Prolog!

The Mints transformation

- Grigori Mints has proven, in his seminal paper studying complexity classes for intuitionistic propositional logic that a formula f is equiprovable to a formula of the form $X_f \to g$ where X_f is a conjunction of formulas of one of the forms $p, \tilde{}p,\ p \to q,\ (p \to q) \to r,\ p \to (q \to r),\ p \to (q\ v\ r),\ p \to \tilde{}q,\ \tilde{}q \to p$
- with introduction of new variables (like in the Tseitin transformation for SAT or ASP solvers), the Mints transformation is linear in space
- we have implemented a variant of the Mints transformation `https://github.com/ptarau/TypesAndProofs/blob/master/mints.pro`
- it also eliminates negation by replacing `~p` with `p->false` and expands the equivalence relation "`<->`"

Applying the Mints transformation

- the correctness of our implementation has been tested by showing that on formulas of small sizes, a trusted prover succeeds on the same set of formulas before and after the transformation
- as transforming formulas known-to-be-true results in formulas of a larger size, we have used them as scalability tests for the provers
- for disjunction-free formulas, in combination with a converter to Nested Horn Clause form, the transformation has been used to generate equivalent Nested Horn Clauses of depth at most 3
- this a new canonical form, also useful for scalability tests for our provers
- note: we turn e.g., `a & b & c -> g` into `a->b->c->g`

```
?- mints(((a & b) -> (c v d)),R).
R = ((a->b->nv2)->(c->nv3)->(d->nv3)->(nv1->nv2->nv3)->(nv2->a)->
     (nv2->b)->(nv3->c v d)->((nv2->nv3)->nv1)->nv1
```

Catching bugs by hardening formulas known-to-be tautologies with the Mints transformation

- before:
  - known to be tautologies in implicational fragment of IPC via the Curry-Howard correspondence
  - formula in Full IPC proven or disproven (by the same prover or other known to be correct prover), before applying the transformation
- after:
  - harder to prove, significantly larger formulas
  - unsound: if proves non-tautology
  - incomplete: if fails to prove known-to-be tautology

A case study: the **fcube 4.1** prover

Donald Knuth: *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

- fcube: a very nice prover by Guido Fiorino, outperforms everything else on the ILTP human-made tests
- version 4.1 at `http://www2.disco.unimib.it/fiorino/fcube.html`
- correctness of underlying calculus proven
- passes all ILTP tests (except for a few timeouts)
- passes all our tests on formulas up to size 12

But testing against the Mints transform finds incompleteness bugs:

```
?- small_taut_bug(4,fcube).
unexpected_failure_on
0->1->2->3->0
<=>
(nv1->0->nv2)->(nv2->1->nv3)->(nv3->2->nv4)->(nv4->3->0)->
    ((0->nv2)->nv1)->((1->nv3)->nv2)->((2->nv4)->nv3)->
    ((3->0)->nv4)->nv1
```

Catching bugs using more general IPC formulas

- we can catch a bug if the "suspect" disagrees with itself on the small easy formula and its hard transform
- we can use agreement with a trusted prover running on the small formula

```
mints_fcube(A):-mints(A,MA),fcube(MA).

?- gold_eq_neg_test(5,mints_fcube,Culprit,Unexpected).
Culprit = ~ (0<->(1<-> ~ (1<->0))), Unexpected = wrong_failure ;
Culprit = ~ (0<->(1<-> ~ (0<->1))), Unexpected = wrong_failure ;
...
```

- `gold_eq_neg_test` compares behavior of a given prover against a trusted "gold standard" prover
- we only display the source, before the transformation is applied
- here we use formulas containing negation and equivalence
- the "prover" consisting of the Mints transform and the suspect fails the test

Transforming to the disjunction-biconditional-negation base

```
toDisjBiCond((A->B),R):-!,toDisjBiCond(A,X),toDisjBiCond(B,Y),
          R=((X v Y)<->Y).
toDisjBiCond(A & B,R):-!,toDisjBiCond(A,X),toDisjBiCond(B,Y),
          R=((X v Y)<->(X<->Y)).
toDisjBiCond(A v B,R):-!,toDisjBiCond(A,X),toDisjBiCond(B,Y),
          R=(X v Y).
toDisjBiCond(A<->B,R):-!,toDisjBiCond(A,X),toDisjBiCond(B,Y),
          R=(X<->Y).
toDisjBiCond(~A,R):-!,toDisjBiCond(A,X),
         R = (~X).
toDisjBiCond(A,A).
```

- this makes formulas larger and much harder to solve, especially as biconditional "<->" is expanded to a conjunction of implications
- a reverse alternative actually works as a good simplifier

A sampling of generators for fragments of IPC and generators restricted to one formula per equivalence class

Nested Horn Clause tree-skeleton generator

```
% OEIS A000108 Catalan 1,2,5,14,42,132,429,1430,4862,16796
genHorn(N,Tree,Leaves):-genHorn(Tree,N,0,Leaves,[]).

genHorn(V,N,N)-->[V].
genHorn((A:-[B|Bs]),SN1,N3)-->{succ(N1,SN1)},[A],
  genHorn(B,N1,N2),
  genHorns(Bs,N2,N3).

genHorns([],N,N)-->[].
genHorns([B|Bs],SN1,N3)-->{succ(N1,SN1)},genHorn(B,N1,N2),
  genHorns(Bs,N2,N3).

?- genHorn(3,H,Vs).
H =  (A:-[B, C, D]), Vs = [A, B, C, D] ;
H =  (A:-[B,  (C:-[D])]), Vs = [A, B, C, D] ;
H =  (A:-[(B:-[C]), D]), Vs = [A, B, C, D] ;
H =  (A:-[(B:-[C, D])]), Vs = [A, B, C, D] ;
H =  (A:-[(B:-[(C:-[D])])]), Vs = [A, B, C, D] .
```

Depth-limited Nested Horn Clause Generators

- all Horn formulas with bodies in canonical order to break symmetries irrelevant for testing provers of depth at most 3
- deeper ones can be reduced to these

```
?- allSortedHorn3(3,T).
T =  (0:-[0,  (0:-[0])]) ;
T =  (0:-[(0:-[(0:-[0])])]) ;
T =  (0:-[1,  (0:-[0])]) ;
T =  (0:-[(1:-[(0:-[0])])]) ;
T =  (0:-[0,  (1:-[0])]) ;
T =  (0:-[(0:-[(1:-[0])])]) ;
.....
T =  (0:-[(1:-[(2:-[2])])]) ;
T =  (0:-[1, 2, 3]) ;
T =  (0:-[1,  (2:-[3])]) ;
T =  (0:-[(1:-[2, 3])]) ;
T =  (0:-[(1:-[(2:-[3])])]) .
```

Implicational Hereditary Harrop Formula Generators

a superset of these is used in the λ-Prolog language

```
harrop_definite(N,Form,Vs):-harrop_definite(Form,Vs,[],N,0).
harrop_goal(N,Form,Vs):-harrop_goal(Form,Vs,[],N,0).

harrop_definite((G->V),[V|Vs1],Vs2)-->harrop_goal(G,Vs1,Vs2).

harrop_goal(V,[V|Vs],Vs)-->[].
harrop_goal((V->G),[V|Vs1],Vs2)-->pred,harrop_goal(G,Vs1,Vs2).
harrop_goal(((H->V)->G),[V|Vs1],Vs3)-->pred,pred,
  harrop_goal(H,Vs1,Vs2),
  harrop_goal(G,Vs2,Vs3).

?- allHarropFormulas(3,T).
T =  (0->0->0->0) ;
...
T =  (((1->1)->0)->2) ;
T =  (((1->2)->0)->2) ;
T =  (((1->2)->0)->3) .
```

Sorted formula generators

- associativity, commutativity, idempotence of conjunction and disjunction
- reflexivity of implication and biconditional
- sorting with duplicate removal eliminates equivalent formulas $\rightarrow$ smaller test sets for given size

Provable formula generators

- formulas of a given size that are provable
- generators for several fragments of IPC filtered by a given prover
- all implicational formulas
- formulas corresponding to a subset of `~`, `->`, `&`, `v`, `<->`
- provable formulas for depth-limited fragments (e.g., results of the Mints transformation or nested Horn clauses of depth at most 3)

Generators for “uninhabitables”

trees that have no inhabitants for all partitions labeling their leaves

```
unInhabitableTree(N,T):-
  genSortedHorn(N,T,Vs),
  \+ (
    natpartitions(Vs),
    hprove(T)
  ).
```

leaf labelings such that no tree they are applied to, has inhabitants

```
unInhabitableVars(N,Vs):-N>0,
  N1 is N-1,
  vpartitions(N,Vs),natvars(Vs),
  \+ (
    genSortedHorn(N1,T,Vs),
    hprove(T)
  ).
```

dual concepts: Motzkin trees that when labeled with any de Bruijn indices result in untypable terms, or binary trees untypable with any S,K labelings

Formula count sequences for small sizes - some in OEIS

- countHornTrees = A000108: Catalan numbers 1,2,5,14,42,132,429,1430,4862,16796
- countSortedHorn = A105633: 1,2,4,9,22,57,154,429,1223,3550,10455,31160,93802,284789
- countHorn3 = NEW: 1,1,2,5,13,37,109,331,1027,3241,10367,33531,109463
- countSortedHorn3=NEW: 1,2,4,8,20,47,122,316,845,2284,6264,17337,48424,136196,385548
- all implicational IPC formulas = A289679: 1, 2, 10, 75, 728, 8526, 115764, 1776060, 30240210
- all provable implicational IPC formulas = NEW: 0,1,3,24,201,2201,27406,391379,6215192
- countUnInhabitableTree = NEW: 1,0,1,1,4,7,23,53,163,432,1306
- countUnInhabitableVars = NEW: 0,1,1,4,9,30,122,528,2517,12951,71455

Generators for random terms and formulas

Random simply-typed terms, with Boltzmann samplers

- we generate random simply-typed normal forms, using a Boltzmann sampler (terms defined with a "natural size" for de Bruijn indices) code is at: `https://github.com/ptarau/TypesAndProofs/blob/master/ranNormalForms.pro`

```
?- ranTNF(60,XT,TypeSize).
XT = l(l(a(a(0, l(a(a(0, a(0, l(...))), s(s(0))))),
          l(l(a(a(0, a(l(...), a(..., ...))), l(0)))))))
      :
      (A->(((A->A)- ...)->D)->D)->M)->M),
TypeSize = 34.
```

Random tautologies from Rémy's algorithm for binary trees labeled with combinators

- Rémy's algorithm (we use Knuth's very efficient algorithm **R**)
- SK-combinator trees + type inference
- X-combinator trees + type inference

Rémy's algorithm+labeling leaves with S,K+type inference

The predicate `ranSK/3` filters the random `SK`-trees of size `N` to represent typable combinator expressions, while ensuring that their types are of size at least `M`, to avoid the frequently occurring trivial types.

```
ranSK(N,M,T):-
  repeat,
    remy_sk(N,X), % generates a binary tree with S or K at its leaves
    sk_type_of(X,T), % infers the type of an SK-expression
    tsize(T,S), % computes the size of the inferred type
    S>=M,
  !,
  natvars(T). % binds type variables to natural numbers, starting from
```

Random implicational formulas from binary trees and set partitions

- The combined generator, produces in a few seconds terms of size 1000:

```
?- ranImpFormula(20,F).
F =  (((0->(((1->2)->1->2->2)->3)->2)->4->(3->3)->
              (5->2)->6->3)->7->(4->5)->(4->8)->8) .

?- time(ranImpFormula(1000,_)).
% includes tabling large Stirling numbers
% 37,245,709 inferences, 7.501 CPU in
7.975 seconds (94% CPU, 4965628 Lips)

?- time(ranImpFormula(1000,_)). % fast, thanks to tabling
% 107,163 inferences, 0.040 CPU in
0.044 seconds (92% CPU, 2659329 Lips)
```

- very fast growth with N, Catalan(N)*Bell(N+1)

Type inference for SK-expressions

The type inference algorithm for SK-expressions is quite simple. After stating that s and k leaves are well typed, we ensure that the types of application nodes agree (as in the modus-ponens rule), using sound unification to avoid creation of cyclical type formulas.

```
sType((A->B->C)->(A->B)->A->C).

kType((A->_B->A)).

sk_type_of(s,T):-sType(T). % S-leaf's type
sk_type_of(k,T):-kType(T). % K-leaf's type
sk_type_of((A*B),Target):- % application node
  sk_type_of(A,SourceToTarget),
  sk_type_of(B,Source),
  unify_with_occurs_check(SourceToTarget,(Source->Target)).
```

Scalability of random typable SK-terms

Large random implicational tautologies, comparable to those generated using Boltzmann samplers, can be produced in a few seconds by inferring the types of random `SK`-expressions.

```
?- ranSK(60,40,T).
T =  (((((0->((1->2->3)->(1->2)->1->3)->4)->0)->0->((1->2->3)->
 (1->2)->1->3)->4)->(0->((1->2->3)->(1->2)->1->3)->4)->0)->
 (((0->((1->2->3)->(1->2)->1->3)->4)->0)->0->
 ((1->2->3)->(1->2)->1->3)->4)->
 ((1->2->3)->(1->2)->1->3)->4).
```

Deriving sound and complete provers

Roy Dyckhoff's **LJT** calculus (implicational fragment)

- termination proven using multiset orderings
- no need for loop checking
- efficient and simple

- LJT_1 : $\dfrac{}{A,\Gamma \vdash A}$
- LJT_2 : $\dfrac{A,\Gamma \vdash B}{\Gamma \vdash A{\rightarrow}B}$
- LJT_3 : $\dfrac{B,A,\Gamma \vdash G}{A{\rightarrow}B,A,\Gamma \vdash G}$ $[A\ atomic\ ]$
- LJT_4 : $\dfrac{D{\rightarrow}B,\Gamma \vdash C{\rightarrow}D \qquad B,\Gamma{\rightarrow}G}{(C{\rightarrow}D){\rightarrow}B,\Gamma \vdash G}$

to support negation, a rule for the special term *false* is needed

- LJT_5 : $\dfrac{}{false,\Gamma \vdash G}$

A prover based on Roy Dyckhoff's LJT calculus, literally

```
lprove(T):-ljt(T,[]),!.

ljt(A,Vs):-memberchk(A,Vs),!.       % LJT_1

ljt((A->B),Vs):-!,ljt(B,[A|Vs]).    % LJT_2

ljt(G,Vs1):- %atomic(G),            % LJT_3
  select((A->B),Vs1,Vs2),
  atomic(A),
  memberchk(A,Vs2),
  !,
  ljt(G,[B|Vs2]).

ljt(G,Vs1):-                        % LJT_4
  select( ((C->D)->B),Vs1,Vs2),
  ljt((C->D), [(D->B)|Vs2]),
  !,
  ljt(G,[B|Vs2]).
```

sprove: extracting the proof terms

```
sprove(T,X):-ljs(X,T,[]).

ljs(X,A,Vs):-memberchk(X:A,Vs),!. % leaf variable
ljs(l(X,E),(A->B),Vs):-!,ljs(E,B,[X:A|Vs]).  % lambda term
ljs(E,G,Vs1):-
  member(_:V,Vs1),head_of(V,G),!, % fail if non-tautology
  select(S:(A->B),Vs1,Vs2),    % source of application
  ljs_imp(T,A,B,Vs2),          % target of application
  !,
  ljs(E,G,[a(S,T):B|Vs2]).     % application

ljs_imp(E,A,_,Vs):-atomic(A),!,memberchk(E:A,Vs).
ljs_imp(l(X,l(Y,E)),(C->D),B,Vs):-ljs(E,D,[X:C,Y:(D->B)|Vs]).

head_of(_->B,G):-!,head_of(B,G).
head_of(G,G).
```

Example: extracting **S**, **K** and **I** from their types

```
?- sprove(((0->1->2)->(0->1)->0->2),X).
X = l(A, l(B, l(C, a(a(A, C), a(B, C))))).                      % S

?- sprove((0->1->0),X).
X = l(A, l(B, A)).                                              % K

?- sprove((0->0),X).                                            % I
X = l(A, A).
```

Hudelmaier's $O(n \, log(n))$ Space prover, restricted to the implicational fragment of IPC

```
nvprove(T):-ljnv(T,[],10000,_).

ljnv(A,Vs)-->{memberchk(A,Vs)},!.
ljnv((A->B),Vs)-->!,ljnv(B,[A|Vs]).
ljnv(G,Vs1)--> % atomic(G),
  {select((A->B),Vs1,Vs2)},
  ljnv_imp(A,B,Vs2),
  !,
  ljnv(G,[B|Vs2]).
                                                  %%        %%   %%
ljnv_imp((C->D),B,Vs)-->!,newvar(P),ljnv(P,[C,(D->P),(P->B)|Vs]).
ljnv_imp(A,_,Vs)-->{memberchk(A,Vs)}.

newvar(N,N,SN):-succ(N,SN).
```

A nested Horn Clause prover, partially evaluated

```
ahprove(A):-toAHorn(A,H),call(H). % this metacall activates '<-'

A<-Vs:-memberchk(A,Vs),!.
(B<-As)<-Vs1:-!,append(As,Vs1,Vs2),B<-Vs2.
G<-Vs1:-                         % atomic(G), G not on Vs1
  memberchk((G<-_),Vs1),  % if not, we just fail
  select(B<-As,Vs1,Vs2),  % outer select loop
  select(A,As,Bs),        % inner select loop
  ahlj_imp(A,B,Vs2),      % A element of the body of B
  !,
  atrimmed(B<-Bs,NewB),   % trim empty bodies
  G<-[NewB|Vs2].
                              %%          %%
ahlj_imp(D<-Cs,B,Vs):-!, (D<-Cs)<-[B<-[D]|Vs].
ahlj_imp(A,_B,Vs):- memberchk(A,Vs).

atrimmed(B<-[],R):-!,R=B.
atrimmed(BBs,BBs).
```

What's *new* with the nested Horn clause form?

- we bypass intermediate steps, by focusing on the head of the Horn clause, which corresponds to the last atom in a chain of implications
- it removes a clause `B:-As` and it removes from its body `As` a formula `A`, to be passed to `ljh_imp`, with the remaining context
- we closely mimic rule LJT_4 by trying to prove `A = (D<-Cs)`, after extending the context with the assumption `B<-[D]`.
- but here we relate `D` with the head `B` !
- the context gets smaller as `As` does not contain the `A` anymore
- if the body `Bs` is empty, the clause is downgraded to its head
- `ahprove` improves execution time compared to `nvprove/1` from **10.98 seconds** down to **4.51 seconds** on terms of size **14** and to **286.836** vs. **95.006** on terms of size **16**
- Can it be that, like with Hudelmaier's optimization, we need only $O(n\ log(n))$ space?

The story behind the $O(n\ log(n))$ space complexity

- we have observed in the past that the Nested Horn Clause prover `hprove/1` outperforms other provers by an order of magnitude (e.g., **121.006** seconds vs. **3221.227** seconds on terms of size **16**).
- we have not had a convincing explanation why this is the case ...
- Hudelmaier's introduction of auxiliary variables brought our implication-based prover much closer in performance to the Nested Horn Clause transform
- does Hudelmaier's optimization shares a relevant similarity with our Nested Horn Clause prover?
- *yes, the duplicated formula* `D` *in* `ahlj_imp/3`*, as it occurs as the head of a clause, is atomic in the Nested Horn Clause prover!*
- $\Rightarrow$ *space increase is bounded by the number of atoms in the original formula to be proven, without the need for introducing new variables*

A Lightweight Theorem Prover for Full Intuitionistic Propositional Logic

the LJT/G4ip sequent calculus for the full IPC + rules for "<->":

```
ljfa(T):-  ljfa(T,[]).

ljfa(A,Vs):-memberchk(A,Vs),!.
ljfa(_,Vs):-memberchk(false,Vs),!.
ljfa(A<->B,Vs):-!,ljfa(B,[A|Vs]),ljfa(A,[B|Vs]).
ljfa((A->B),Vs):-!,ljfa(B,[A|Vs]).
ljfa(A & B,Vs):-!,ljfa(A,Vs),ljfa(B,Vs).
ljfa(G,Vs1):- % atomic or disj or false
  select(Red,Vs1,Vs2),
  ljfa_reduce(Red,G,Vs2,Vs3),
  !,
  ljfa(G,Vs3).
ljfa(A v B, Vs):-(ljfa(A,Vs);ljfa(B,Vs)),!.
```

continued

```
ljfa_reduce((A->B),_,Vs1,Vs2):-!,ljfa_imp(A,B,Vs1,Vs2).
ljfa_reduce((A & B),_,Vs,[A,B|Vs]):-!.
ljfa_reduce((A<->B),_,Vs,[(A->B),(B->A)|Vs]):-!.
ljfa_reduce((A v B),G,Vs,[B|Vs]):-ljfa(G,[A|Vs]).

ljfa_imp((C->D),B,Vs,[B|Vs]):-!,ljfa((C->D),[(D->B)|Vs]).
ljfa_imp((C & D),B,Vs,[(C->(D->B))|Vs]):-!.
ljfa_imp((C v D),B,Vs,[(C->B),(D->B)|Vs]):-!.
ljfa_imp((C<->D),B,Vs,[((C->D)->((D->C)->B))|Vs]):-!.
ljfa_imp(A,B,Vs,[B|Vs]):-memberchk(A,Vs).
```

While not the fastest prover or the most flexible handling hard human-made tests, this is so far, the only Prolog-based prover that is sound, complete and safe from space explosions.

Conclusions and future work

- cross-testing opportunities:
  - type inference algorithms for lambda terms and combinator expressions and theorem provers for propositional intuitionistic logic
  - a "virtuous circle": transformers help debug provers and provers help debug transformers
- the derived lightweight provers:
  - more likely than provers using complex heuristics to be sound and complete
  - also, more they can be more easily turned into parallel implementations
  - provers working on nested Horn clauses outperform those working directly on implicational formulas
  - now we cover full intuitionistic propositional logic
- future work
  - formally describe the nested Horn-clause prover in sequent-calculus
  - explore compilation techniques and parallel algorithms
  - a work on a generalization to nested Horn clauses with conjunctions and universally quantified variables and grounding techniques as used by SAT and ASP solvers