Combinatorial Testing Techniques for Propositional Intuitionistic Theorem Provers

Paul Tarau

University of North Texas

CLA'2018

Research supported by NSF grant **1423324**

Outline

code is available at: **`https://github.com/ptarau/TypesAndProofs`**

The implicational fragment of propositional intuitionistic logic

Hilbert-style axioms schemes for the implicational fragment of propositional intuitionistic logic

the implicational fragment of intuitionistic propositional logic can be defined by two axiom schemes:

- $K: \quad A \to (B \to A)$
- $S: \quad (A \to (B \to C)) \to ((A \to B) \to (A \to C))$

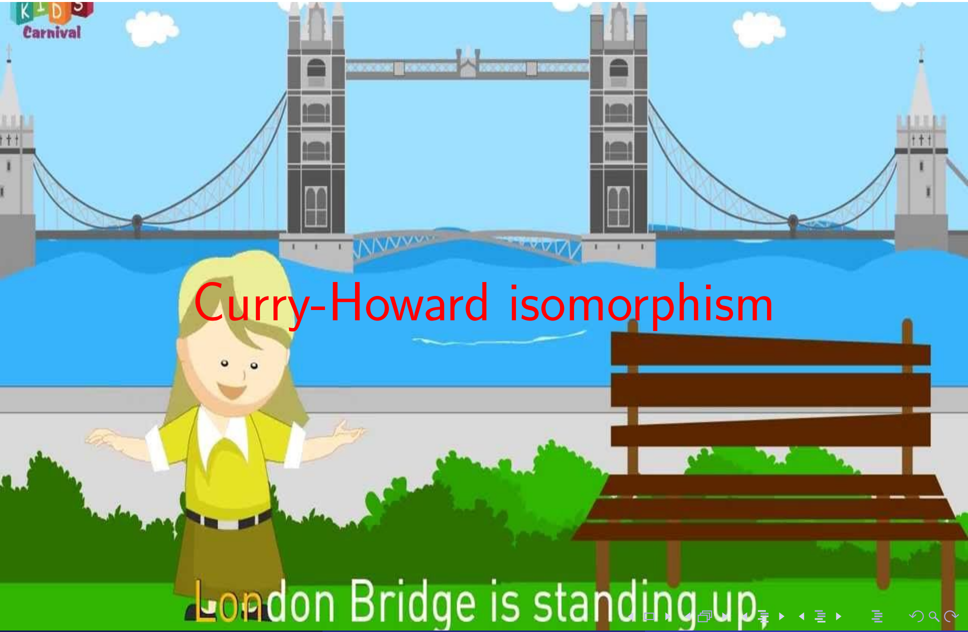
and the modus ponens inference rule:

- $MP: \quad A,\ A \to B\ \vdash\ B.$
- substitution

The insight: *those are exactly the types of the combinators* **S** *and* **K**!

Is there a bridge standing up between the two sides?

The bridge between **types** and **propositions**: standing up!



Curry-Howard isomorphism

The Curry-Howard isomorphism

it connects:

- the implicational fragment of propositional intuitionistic logic
- types in the *simply typed lambda calculus*

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

$\Rightarrow$

- lambda term (typically in normal form) can serve as a witness for the existence of a proof for the corresponding tautology in minimal logic
- a theorem prover can also be seen as a tool for program synthesis

Proof systems for intuitionistic implicational propositional logic

Gentzen's **LJ** calculus, reduced to the implicational fragment of intuitionistic propositional logic

- LJ_1 : $\dfrac{}{A,\Gamma \vdash A}$
- LJ_2 : $\dfrac{A,\Gamma \vdash B}{\Gamma \vdash A \to B}$
- LJ_3 : $\dfrac{A \to B,\Gamma \vdash A \qquad B,\Gamma \vdash G}{A \to B,\Gamma \vdash G}$

- rules, if implemented directly are subject to looping
- several variants use loop-checking, by recording the sequents used

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

- replace LJ_3 with LJT_3 and LJT_4
- termination proven using multiset orderings
- no need for loop checking
- efficient and simple

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

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

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

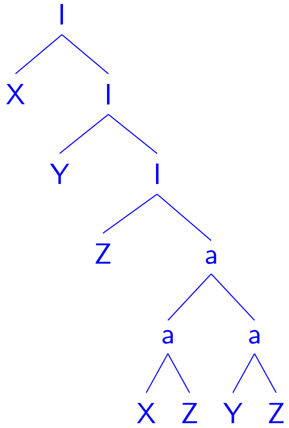
An executable specification

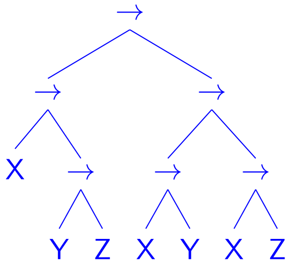
Notations and assumptions

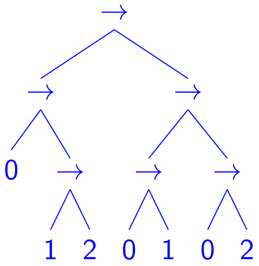
- we use **Prolog** as our meta-language
- code (now grown to above *2000 lines*) at **`https://github.com/ptarau/TypesAndProofs`**
- basic Prolog programming:
  - variables will be denoted with uppercase letters
  - the pure Horn clause subset
  - well-known built-in predicates like `memberchk/2` and `select/3`, `call/N`), CUT and `if-then-else` constructs
- 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`" and *logic variables (or atoms or integers) as their leaves*

Examples

the **S** combinator and its type, with variables and integers as leaves:







The importance of being Leanest

- Roy Dyckchoff's program, about 420 lines
- can we just use his calculus as a starting point?
- a blast from the past: lean theorem provers can be fast!

$\Rightarrow$

- we start with a simple, almost literal translation of rules $LJT_1 \ldots LJT_4$ to Prolog
- note: values in the environment Γ denoted by the variables `Vs, Vs1, Vs2...`

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

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

Deriving our *lean* theorem provers

bprove: concentrating nondeterminism into one place

The first transformation merges the work of the two `select/3` calls into a single call, observing that they do similar things after the call. That avoids redoing the same iteration over candidates for reduction.

```
bprove(T):-ljb(T,[]),!.

ljb(A,Vs):-memberchk(A,Vs),!.
ljb((A->B),Vs):-!,ljb(B,[A|Vs]).
ljb(G,Vs1):-
  select((A->B),Vs1,Vs2),
  ljb_imp(A,B,Vs2),
  !,
  ljb(G,[B|Vs2]).

ljb_imp((C->D),B,Vs):-!,ljb((C->D),[(D->B)|Vs]).
ljb_imp(A,_,Vs):-atomic(A),memberchk(A,Vs).
```

$\Rightarrow$ 51% speed improvement for formulas with 14 internal nodes

Calls for proving **S**

```
?- s_(S),bprove(S).

[]-->(0->1->2)->(0->1)->0->2
[(0->1->2)]-->(0->1)->0->2
[(0->1),(0->1->2)]-->0->2
[0,(0->1),(0->1->2)]-->2
[1,0,(0->1->2)]-->2
[(1->2),1,0]-->2
[2,1,0]-->2
S =  ((0->1->2)->(0->1)->0->2).
```

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).
```

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).

Tamari order:

?- T=(((a->b)->c) -> (a->(b->c))), sprove(T,X).
T =  (((a->b)->c) -> a->(b->c)),
X = l(A, l(B, l(C, a(A, l(D, l(E, C)))))).

?- T=((a->(b->c)) -> ((a->b)->c)), sprove(T,X).
false.
```

Inferring **S** from its type

```
?- s_(S),sprove(S,X),nv(X).
[]-->A:((0->1->2)->(0->1)->0->2)
[A:(0->1->2)]-->B:((0->1)->0->2)
[A:(0->1),B:(0->1->2)]-->C:(0->2)
[A:0,B:(0->1),C:(0->1->2)]-->D:2
[a(A,B):1,B:0,C:(0->1->2)]-->D:2
[a(A,B):(1->2),a(C,B):1,B:0]-->D:2
[a(a(A,B),a(C,B)):2,a(C,B):1,B:0]-->D:2

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

Implicational formulas as embedded Horn Clauses

- equivalence between:
  - $B_1 \rightarrow B_2 \ldots B_n \rightarrow H$ and
  - H :- $B_1, B_2, \ldots, B_n$ (in Prolog notation)
- *H* is the *atomic* formula ending a chain of implications
- we can recursively transform an implicational formula:

```
toHorn((A->B),(H:-Bs)):-!,toHorns((A->B),Bs,H).
toHorn(H,H).

toHorns((A->B),[HA|Bs],H):-!,toHorn(A,HA),toHorns(B,Bs,H).
toHorns(H,[],H).
```

- the transformation is reversible!

```
?- toHorn(((0->1->2)->(0->1)->0->2),R).
R =  (2:-[(2:-[0, 1]),  (1:-[0]), 0]).

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

Transforming provers for implicational formulas into equivalent provers working on embedded Horn clauses

```
hprove(T0):-toHorn(T0,T),ljh(T,[]),!.

ljh(A,Vs):-memberchk(A,Vs),!.
ljh((B:-As),Vs1):-!,append(As,Vs1,Vs2),ljh(B,Vs2).
ljh(G,Vs1):-                  % atomic(G), G not on Vs1
  memberchk((G:-_),Vs1),      % if non-tautology, we just fail
  select((B:-As),Vs1,Vs2),    % outer select loop
  select(A,As,Bs),            % inner select loop
  ljh_imp(A,B,Vs2),           % A is in the body of B
  !,trimmed((B:-Bs),NewB),    % trim empty bodies
  ljh(G,[NewB|Vs2]).

ljh_imp(A,_B,Vs):-atomic(A),!,memberchk(A,Vs).
ljh_imp((D:-Cs),B,Vs):- ljh((D:-Cs),[(B:-[D])|Vs]).

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

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

The *embedded Horn clause form helps bypassing some intermediate steps, by focusing on the head of the Horn clause, which corresponds to the last atom in a chain of implications*. Also, 69% faster on terms of size 15.

- the 3-rd clause of `ljh` works as a context reducer
- a second `select/3` call in it gives `ljh_imp` more chances to succeed and commit
- 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
- if `A` is atomic, we succeed if and only if it is already in the 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

A lifting to classical logic, via Glivenko's transformation

Glivenko's translation that prefixes a formula with its double negation. It turns an intuitionistic propositional prover into a classical one.

- we add the atom *false*, to the language of the formulas
- we rewrite negation of x into $x \rightarrow false$
- we add the special handling of `false` as the first clause of the predicate `ljb/2`, corresponding to rule

$$LJT_5: \quad \frac{}{false, \Gamma \vdash G}$$

```
ljb(_AnyGoal,Vs):-memberchk(false,Vs),!.
```

The 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
  - indirect testing: via Glivenko's translation
  - soundness and completeness are 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 all lambda terms up to a given size: types of simply typed lambda terms are tautologies for sure
- exhaustive vs. random
  - all implicational formulas up to given size: a mix of non-tautologies and tautologies (fewer and fewer with size)
  - type of all lambda terms of a given size, random simply typed terms
  - random simply typed lambda terms, random implicational formulas

Finding false negatives by generating the set of simply typed normal forms of a given size

search is by the size of the proof rather than the size of the formula:

- a false negative is identified if our prover fails on a type expression known to have an inhabitant
- via the Curry-Howard isomorphism, such terms are the types inferred for lambda terms, generated by increasing sizes
- this means that implicational formulas having proofs shorter than a given number are all covered
- but possibly small formulas having long proofs might not be reachable
- code for generating all simply typed terms at: `https://github.com/ptarau/TypesAndProofs/blob/master/allTypedNFs.pro`

Examples of simply typed normal forms and their types

```
?- tnf(4,X:T),nv(T+X).
X = l(E, l(F, l(G, l(H, H)))),
T =  (A->B->C->D->D) ;
X = l(E, l(F, l(G, l(H, G)))),
T =  (A->B->C->D->C) ;
X = l(E, l(F, l(G, l(H, F)))),
T =  (A->B->C->D->B) ;
X = l(E, l(F, l(G, l(H, E)))),
T =  (A->B->C->D->A) ;
X = l(C, l(D, a(D, C))),
T =  (A->(A->B)->B) ;
X = l(C, l(D, a(C, D))),
T =  ((A->B)->A->B) ;
X = l(C, a(C, l(D, D))),
T =  (((A->A)->B)->B) .
```

- generation and type inference are interleaved
- ⇒ we can scale up to size 20 (with l=1,a=2 size definition)

Finding false positives by generating all implicational formulas/type expressions of a given size

- a false positive is identified if the prover succeeds finding an inhabitant for a type expression that does not have one
- generating all implicational formulas of a given size:
  - generating all binary trees of a given size
  - extracting their leaf variables
  - iterating over the set of the set partitions of all leaves, while unifying variables belonging to the same partition
- the code describing the all-tree and all-set partition generation, as well as their integration as a type expression generator, is at: `https://github.com/ptarau/TypesAndProofs/blob/master/allPartitions.pro`.

Examples of implicational formulas

```
?- showImpForms(2).
A->A->A
A->B->A
A->A->B
A->B->B
A->B->C
(A->A)->A
(A->B)->A
(A->A)->B
(A->B)->B
(A->B)->C
```

- we use set partitions of `{A,B,C}` to "name" the variables
- $\Rightarrow$ we generate them up to α-equivalence

Sizes of our test data for all term generators

- simply typable lambda terms in normal forms
  - size definition variables=0, lambdas=1, applications=2
  - up to 20
  - 0, 1, 2, 3, 7, 17, 43, 129, 389, 1245, 4274, 14991, 55289, 210743,826136, 3354509, 13948176, 59553717, 260593082, 1164467603, **5,321,739,900**, ...
- implicational formulas=types
  - A289679: Catalan(n-1)*Bell(n)
  - up to 10
  - 1, 2, 10, 75, 728, 8526, 115764, 1776060, 30240210, **563,870,450**

Trimming out symmetries in implicational formulas

- as both Catalan numbers and Bell numbers grow very fast, we need to trim our test generator to smaller equivalence classes
- step 1: by having set partitions instead of all combinations (done)
- step 2: as the conjunction in the body of $H :- B_1, B_2, \ldots, B_n$ is associative, commutative and B_i, B_i is equivalent to B_i we can see $B_1, B_2, \ldots, B_n$ as a set rather then a sequence
- $\Rightarrow$ we recursively trim as we generate, with all formulas in strictly increasing order and duplicates trimmed out
- we have significantly fewer than Catalan(n-1)*Bell(n)

```
?- ncounts(8,allImpFormulas(_,_)). % without trimming
counts=[1,2,10,75,728,8526,115764,1776060,30240210]
ratios=[2,5,7.5,9.7,11.71,13.57,15.34,17.02]

?- ncounts(8,allSortedHorn(_,_)). % trimmed. OPEN QUEST: ANALYTICS?
counts=[1,2,7,38,266,2263,22300,247737,3049928]
ratios=[2,3.5,5.42,7,8.5,9.85,11.1,12.31]
```

Testing against a trusted reference implementation

Once we can trust an existing reference implementation (e.g., after it passes our generator-based tests), it makes sense to use it as a gold standard. Thus, we can identify both false positives and negatives directly!

```
gold_test(N,Generator,Gold,Silver, Term, Res):-
  call(Generator,N,Term),
  gold_test_one(Gold,Silver,Term, Res),
  Res\=agreement.

gold_test_one(Gold,Silver,T, Res):-
  ( call(Silver,T) -> \+ call(Gold,T),
    Res = wrong_success
  ; call(Gold,T) -> % \+ Silver
    Res = wrong_failure
  ; Res = agreement
  ).
```

Examples of *gold standard*-based tests

we can use a generator for all implicational formulas, and Dyckhoff's `dprove/1` predicate as a gold standard:

```
gold_test(N, Silver, Culprit, Unexp):-
  gold_test(N,allImpFormulas,dprove,Silver,Culprit,Unexp).
```

to "test the tester", we design a prover that randomly succeeds or fails:

```
badProve(_) :- 0 =:= random(2).

?- gold_test(6,lprove,T,R).
false. % indicates that no false positive or negative is found

?- gold_test(6,badProve,T,R).
T =  (0->1->0->0->0->0->0),
R = wrong_failure ;
...
?- gold_test(6,badProve,T,wrong_success).
T =  (0->1->0->0->0->0->2) ;
T =  (0->0->1->0->0->0->2) ;
...
```

Catching unsoundness

```
badSolve(A,Vs):-atomic(A),!,memberchk(A,Vs).
badSolve((A->B),Vs):-badSolve(B,[A|Vs]).
badSolve(_,Vs):-badReduce(Vs).

badReduce([]):-!.
badReduce(Vs):-select(V,Vs,NewVs),badSolve(V,NewVs),badReduce(NewVs).
```

- a more interesting case is when a prover is only guilty of false positives
- naively implementing an apparently sound intuition:
  - a goal is provable w.r.t. a context `Vs` if all its premises are provable
  - with implication introduction assuming premises
  - success achieved when the environment is reduced to empty

```
?- gold_test(6,badSolve,T,wrong_failure).
false. % no wrong failure

?- gold_test(6,badSolve,T,wrong_success).
T =  (0->0->0->0->0->0->1) ;
T =  (0->1->0->0->0->0->2) ;
...
```

Testing on classical formulas, via Glivenko's translation

- an implicit correctness test: we compare the behavior of a prover that handles `false`, with Glivenko's double negation transformation
- we turn our intuitionistic propositional provers into a classical provers
- we work on classical formulas containing implication and negation operators

```
gold_classical_test(N,Silver,Culprit,Unexpected):-
  gold_test(N,allClassFormulas,tautology,Silver,
  Culprit,Unexpected).
```

- we are using Melvin Fitting's classical tautology prover `tautology/1` as a gold standard
- we restricted to implicational logic, see: `https://github.com/ptarau/TypesAndProofs/blob/master/third_party/fitting.pro`.

Examples of tests on classical formulas

- we modify our generators to use `false` instead of one of our variables in the all implicative formula generator

```
?- gold_classical_test(7,gprove,Culprit,Error).
false. % no false positive or negative found

?- gold_classical_test(7,kprove,Culprit,Error).

Culprit =  ((false->false)->0->0->((1->false)->false)->1),
Error = wrong_failure ;

Culprit =  ((false->false)->0->1->((2->false)->false)->2),
Error = wrong_failure .
...
```

- `gprove/1`, implementing Glivenko's translation, passes the test
- `kprove/1`, that handles only intuitionistic tautologies (including negated formulas), will fail on classical tautologies that are not also intuitionistic tautologies

Performance and scalability testing

Can we make a tautology so hard that we cannot prove it?

- After a few refining steps, our best provers solved every problem we threw at them.
- Can we generate harder ones, that they cannot solve?
- For a PSPACE-complete problem, that should be easy!

Figure: Can God make a rock so heavy that He cannot lift it?

A balancing act: the question has its nuances!

- Yes, very likely so, for human made ones, see: `http://www.iltp.de/`.

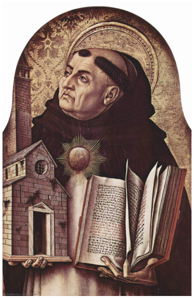


Figure: We can save face by rephrasing the question!

- Can we generate such tautologies automatically?
- Can we generate non-tautologies that we cannot refute?

Random simply-typed terms, with Boltzmann samplers

- we generate random simply-typed normal forms, using a Boltzmann sampler
- the code variant, adapted to our different term-size definition is at: https://github.com/ptarau/TypesAndProofs/blob/master/ranNormalForms.pro
- it works as follows:

```
?- ranTNF(10,XT,TypeSize).
XT = l(l(a(a(s(0), l(s(0))), 0))) : (((A->B)->B->C)->B->C).
TypeSize = 5.

?- 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.
```

Most random lambda terms have types that are too easy!

- there's some variation in the size of the terms that Boltzmann samplers generate
- but the problem occurs despite the fairly large simply typed normal forms we can generate (e.g., above "natural" size=60)
- uniform distribution of normal forms leads often to simple tautologies where an atom identical to the last one is contained in the implication chain leading to it
- $\Rightarrow$ if we want to use these for scalability tests, additional filtering mechanisms need to be devised, to statically reject type expressions that are large, but easy to prove as intuitionistic tautologies
- back to the same challenge:
  Can we make a tautology so hard that we cannot prove it?

Random implicational formulas

- Rémy's algorithm for the *generation of random binary binary trees*: `https://github.com/ptarau/TypesAndProofs/blob/master/RemyR.pro`
- a *random set partition generator*: `https://github.com/ptarau/TypesAndProofs/blob/master/ranPartition.pro`.
- automatic Boltzmann sampler generators for a partition are limited to a fixed numbers of equivalence classes for which a CF- grammar can be given
- $\Rightarrow$ we build our the random set partition generator that groups variables in leaf position into equivalence classes by using an Stam's urn-algorithm: `https://github.com/ptarau/TypesAndProofs/blob/master/ranPartition.pro`

Examples of formula generation

- Thus, the partition generator works as follows:

```
?- ranSetPart(7,Vars).
Vars = [0, 0, 1, 1, 2, 3, 0] .

?- ranSetPart(7,Vars).
Vars = [0, 1, 2, 1, 1, 2, 3] .
```

- we obtain the the binary tree generated by Rémy's algorithm, by unifying the list of variables `Vs` with it

```
?- remy(6,T,Vs).
T =  ((((A->B)->C->D)->E->F)->G),
Vs = [A, B, C, D, E, F, G] .
```

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)
```

.

Testing with large random terms

- testing for false positives and false negatives for random terms: similar to exhaustive testing with terms of a given size
- assuming Roy Dyckhoff's prover as a gold standard, we can find out that our embedded Horn Clause prover `hprove/1` can handle 100 terms of size 50 as well as the gold standard

```
?- gold_ran_imp_test(50,100,hprove, Culprit, Unexpected).
false. % indicates no differences with the gold standard
```

- the size of the random terms handled by `hprove/1` makes using provers an *appealing alternative to random lambda term generators in search for very large lambda term / simple type pairs*.

On the side of random simply typed terms, limitations come from their vanishing density, while on the other side of the Curry-Howard isomorphism they come from the PSPACE-complete proof procedures.

Can our lean provers actually be fast? A quick performance evaluation

- our benchmarking code is at: `https://github.com/ptarau/TypesAndProofs/blob/master/bm.pro`.
- we compare our provers on:
  - known tautologies with given proof size N (lambda terms in normal forms)
  - implicational formulas of size $(N//2)$
  - for the winner, we also test it on larger formulas up to size 20 and 10

Runtimes on known tautologies and all formulas

Prover	Term Size	Positive	Mix (half size)	Total seconds
lprove	13	1.4	0.28	1.68
lprove	14	6.86	6.33	13.2
lprove	15	56.93	6.56	**63.49**
bprove	13	0.92	0.20	1.12
bprove	14	4.31	4.26	8.58
bprove	15	31.72	4.31	**36.03**
sprove	13	1.92	0.16	2.09
sprove	14	9.43	2.72	12.16
sprove	15	48.55	2.73	**51.29**
hprove	13	0.95	0.11	1.07
hprove	14	4.26	1.86	6.12
hprove	15	19.35	1.87	**21.22**
dprove	13	2.18	0.35	2.53
dprove	14	10.96	6.25	17.21
dprove	15	1100.72	5.76	**1106.49**

How does `hprove/1` scale?

Prover	Size	Positive	Mix (half-size)	Total Time
hprove	16	89.58	34.74	124.32
hprove	17	427.47	33.56	461.03
hprove	18	2090.77	684.15	2774.92
hprove	19	11270.35	756.8	**12027.15**

Figure: hprove/1 on larger tests, time in seconds

- no unexpected slowdown on either proving known tautologies or rejecting non-tautologies (contrary to `dprove/1` that gave up already on size `15`)
- small enough to be converted to C, possibly competitive with much more complex provers
- needs to be tested on hard, human-made formulas (e.g., those known to have exponentially long proofs)

A look at parallel algorithms for provers and testers

Finding the first solution in parallel, with backtracking generator

- we have built a custom multi-threading manager, for non-deterministic generators
- message queue-based inter-thread communication: `nondet_first/3`
- thread-pool: 2/3 of the available threads
- workers pull goals provided by nondeterministic generator
- each worker backtracks on its slice of work independently
- first successful worker returns answer and stops all threads
- code at `https://github.com/ptarau/TypesAndProofs/blob/master/nd_threads.pro`

Parallel generation of random implicational tautologies

- we use `nondet_first/3` to lift our sequential implicational formula generator `ranImpFormulas` to a parallel one

```
ran_typed_ground(Seed,PartCount,TreeCount,Prover,X:G):-
  Gen=ranImpFormulas(Seed,PartCount,TreeCount,G),
  Exec=call(Prover,G),
  nondet_first(G,Exec,Gen),
  ljs(X,G),
  !.
```

- once our (possibly faster) `Prover` succeeds, the predicate `ljs/2` is used to find the actual proof in the form of a lambda term
- using Seed=2018, and an implicational formula of size 60 with up to 40 distinct variables, we obtain:

```
?- ran_typed_ground(2018,60,40,bprove,X:G).
X = l(A, l(B, l(C, l(D, l(E, l(F, l(G, C))))))),
G =  ((0->(1->2)->3)->4->5->.. -> ...)->5).
```

Can we generate (automatically) a tautology so hard that we cannot prove or refute it? **YES!**

- we often generate larger tautologies than those obtained as types of lambda terms using Boltzmann samplers, the large number of parallel threads (e.g., iMacPro with 18 cores/36 threads) will favor random implicational formulas having short proofs over those with longer proofs
- so this method can win over tautologies generated with Boltzmann samplers, even if most of these tend to be easier to solve
- but the problem is now harder: prove becomes prove or refute
- as we can push size above 1000 we have easily have some that we cannot solve (prove or refute):

```
?- Seed is random(1000),timed_call(600,
   ran_typed_ground(Seed,100,50,hprove,T),Time).
Seed = 82,
Time = timeout_after(600).
```

Parallel generation of normal forms and their types

- we ensure all parallel threads are driven by a unique random seed so that we can replicate runs
- the actual work is performed by `parRanTypableNF/8` which simply tries as many sequential generators as the number of available threads
-
```
?- parRanTypedTNF(42,50,40,1,X:T,Size).
X = l(A, l(B, a(a(C, a(C, l(D, l(E,...l...)), I), J))))))),
T =  (L->((M->M)->(N->O->N)->P->N->O->N)->...->N->O->N)->
((Q->(M->M)->(N-> ...)->P-> ... -> ...)->P)->P->N->O->N),
Size = 43.
```
- this generates, using the seed 42, a random lambda term in normal form and its type, of (approximately) sizes 40 and 50 respectively in a few seconds

Parallelizing the provers

- given the small granularity of the search component of our provers, our initial attempts did not indicate performance gains
- multiple term copies between message queues turned out to be costlier than the benefits of parallel search with a very large number of very light tasks
- $\Rightarrow$ statically create equivalent term variants so that multiple distinct proof sequences are tried in parallel
- e.g., in $H :- B_1, B_2, ..., B_n$ randomly permute the terms $B_1, B_2 ... B_n$

```
?- ihard(Hard),time(bprove(Hard)).
% 13,152,259,741 inferences, 2215.060 CPU in 2217.478 seconds

?- ihard(Hard),time(parProveHorn(hprove,Hard)).
% 25,442 inferences, 0.012 CPU in 0.684 seconds ...
false. % <= refuted as non-tautology
```

- the formula of size 100 is instantly solved with the parallelized `hprove/1` while it takes more than 2000 seconds with `bprove/1`

Related work I

- the related work derived from Gentzen's **LJ** calculus is in the hundreds if not in the thousands of papers and books
- [1, 2]: our starting points for deriving our provers, directly from the **LJT** calculus
- similar calculi, key ideas of which made it into the Coq proof assistant's code: in [3]
- [4] described in full detail in [5], finds and/or counts inhabitants of simple types in long normal form
- interestingly, these algorithms have not crossed, at our best knowledge, to the other side of the Curry-Howard isomorphism in the form of theorem provers

Related work II

- overviews of closely related calculi, using the implicational subset of propositional intuitionistic logic are [6, 2].
- using hypothetical implications in Prolog, although all with a different semantics than Gentzen's **LJ** calculus or its **LJT** variant, go back as early as [7], followed by a series of Lambda-Prolog and linear logic-related books and papers, e.g., [8]
- the similarity to the propositional subsets of N-Prolog [7] and λ-Prolog [8] comes from their close connection to intuitionistic logic
- but neither derive implementations from a pure **LJ**-based calculus or have termination properties implemented along the lines the **LJT** calculus

Dyckhoff, R.:
Contraction-free sequent calculi for intuitionistic logic.
Journal of Symbolic Logic **57**(3) (1992) 795âĂŞ807

Dyckhoff, R.:
Intuitionistic Decision Procedures Since Gentzen.
In Kahle, R., Strahm, T., Studer, T., eds.: Advances in Proof Theory, Cham, Springer International Publishing (2016) 245–267

Herbelin, H.:
A Lambda-Calculus Structure Isomorphic to Gentzen-Style Sequent Calculus Structure.
In: Selected Papers from the 8th International Workshop on Computer Science Logic. CSL '94, London, UK, UK, Springer-Verlag (1995) 61–75

Ben-Yelles, C.B.:
Type assignment in the lambda-calculus: Syntax and semantics.
PhD thesis, University College of Swansea (1979)

Hindley, J.R.:
Basic Simple Type Theory.
Cambridge University Press, New York, NY, USA (1997)

Gabbay, D., Olivetti, N.:
Goal-oriented deductions.
In: Handbook of Philosophical Logic.
Springer (2002) 199–285

Gabbay, D.M., Reyle, U.:
N-prolog: An extension of prolog with hypothetical implications. i.
The Journal of Logic Programming **1**(4) (1984) 319–355

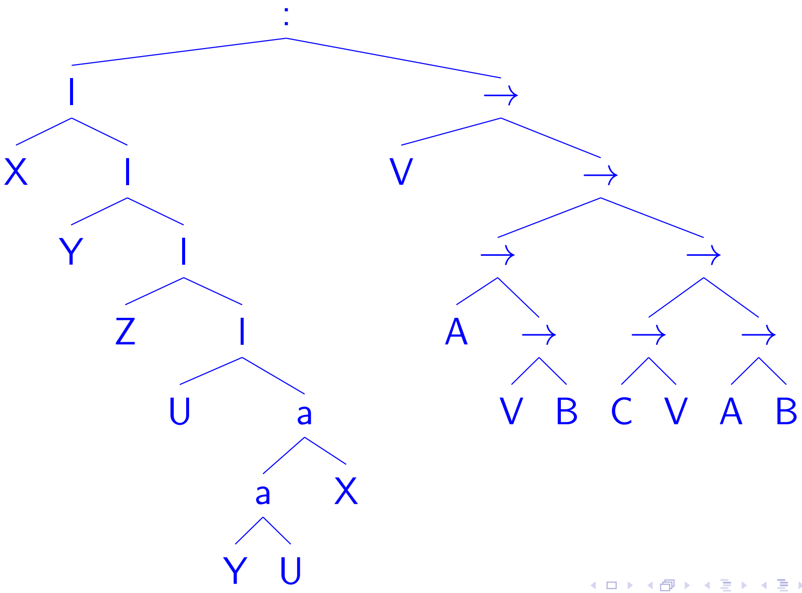
Miller, D., Nadathur, G.:
Programming with Higher-Order Logic.
Cambridge University Press, New York, NY, USA (2012)

Conclusions and future work

Conclusions and future work

- our empirically oriented approach has found variants of lean propositional intuitionistic provers that are comparable to their more complex peers, derived from similar calculi
- besides the derivation of our lean theorem provers, our code base at **`https://github.com/ptarau/TypesAndProofs`** also provides an extensive test-driven development framework built on several cross-testing opportunities between type inference algorithms for lambda terms and theorem provers for propositional intuitionistic logic
- the *embedded Horn clause provers* might be worth *formalizing as a calculus* and subject to deeper theoretical analysis
- extension to full propositional and first order intuitionistic logic seems easy
- given that they share their main data structures with Prolog, it seems interesting to attempt their partial evaluation or compilation to Prolog

Questions?


:
l
→
X
l
V
→
Y
l
→
→
Z
l
A
→
→
→
U
a
V
B
C
V
A
B
a
X
Y
U