A Combinatorial Testing Framework for Intuitionistic Propositional Theorem Provers

Paul Tarau

University of North Texas

PADL'2019

Research supported by NSF grant **1423324**

Overview

- In search for an efficient but minimalist theorem prover:
  - we design a combinatorial testing framework, using
    - known to be provable formulas: types inferred for lambda terms
    - as all-term generators
    - random term generators
- We choose Prolog as our meta-language, because:
  - it reduces the semantic gap (derived from essentially the same formalisms as those we are covering)
  - has the right language constructs for a concise and efficient declarative implementation
- Our implementation is available at: **`https://github.com/ptarau/TypesAndProofs`**.
- we start with a few derivation steps for our provers
- next, we describe the testing framework used to validate these steps

Test-driven derivation steps from proof systems to executable code

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

$\Rightarrow$

- 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

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

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

- rules, if implemented directly are subject to looping
- several variants use loop-checking, by recording the sequents used
- when implementing them in Prolog, we read them backward, the goal to prove is below the line

Roy 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 : $\dfrac{A,\Gamma \vdash B}{\Gamma \vdash A \rightarrow B}$
- 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 : $\dfrac{}{false,\Gamma \vdash G}$

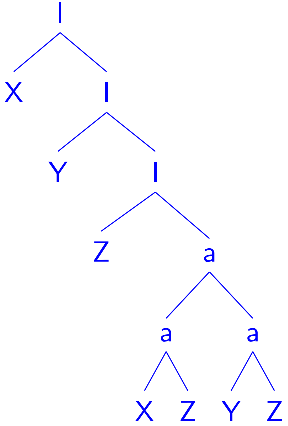
Prolog as a meta-language

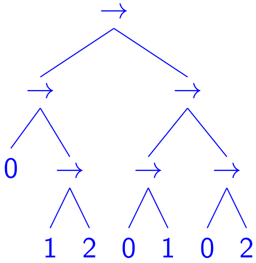
- we use **Prolog** as our meta-language
- Prolog programming background:
  - variables will be denoted with uppercase letters
  - the pure Horn clause subset
  - well-known built-in predicates like `memberchk/2` and `select/3`, `call/N`), CUT and `if-then-else` constructs
- lambda terms: **a/2**=application, **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

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

Examples

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





The first step: from Sequent Calculus to Prolog

- Roy Dyckchoff's program, about 420 lines
- tableau-based provers implementing sophisticated heuristics are often above 1000 lines of code
- $\Rightarrow$ what if we just use the elegant and simple **LJT** calculus as a starting point?
- the simpler a prover is, the easier is to prove formally its correctness
- also, possibly it will be easier to parallelize or implement in a different language

$\Rightarrow$

- 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`....

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]).
```

bprove: concentrating nondeterminism into one place

- we merges the work of the two `select/3` calls into a single call
- they do similar things after the call!

```
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).
```

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).
```

Implicational formulas as nested Horn Clauses

- equivalence between:
  - $B_1 \to B_2 \to \ldots \to B_n \to H$ and
  - $H\ :-\ B_1, B_2, \ldots, B_n$ (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).

?- 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]).
```

- also, note that the transformation is reversible!

Transforming provers for implicational formulas into equivalent provers working on nested 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 nested Horn clause form?

The *nested 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*.

- 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

69% faster on terms of size **15**.

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 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
- all-term vs. random testing
  - all typed terms of a given size, known to be tautologies
  - all implicational formulas up to given size: a mix of non-tautologies and tautologies (fewer and fewer with size)
  - random simply typed lambda terms
  - random implicational formulas

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

- 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 all implicational formulas having proofs shorter than a given number are covered
- but, *small formulas having long proofs* might not be reachable with this method that explores the search by the size of the proof rather than the size of the formula to be proven!

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.
- we obtain type expressions by generating all binary trees of a given size, extracting their leaf variables and then iterating over the set of their set partitions, while unifying variables belonging to the same partition
- code at: `https://github.com/ptarau/TypesAndProofs/blob/master/allPartitions.pro`.
- an advantage of exhaustive testing with all formulas of a given size is that it implicitly ensures coverage: no path is missed simply because there are no paths left unexplored
- but, we need an oracle that tells as which formulas should succeed and which should fail!
- $\Rightarrow$ we need a trusted reference implementation!

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
   ).
```

Random simply-typed terms, with Boltzmann samplers

- once passing correctness tests, our provers need to be tested against large random terms (a scalability test)
- we generate random simply-typed normal forms, using a Boltzmann sampler
  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 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)
```

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

Scalability testing: a quick performance evaluation

- our benchmarking code is at: `https://github.com/ptarau/TypesAndProofs/blob/master/bm.pro`.
- we compare our provers on on a blend of:
  - known tautologies with given proof size N (lambda terms in normal form)
  - implicational formulas of size $(N//2)$

Runtimes on known tautologies and mix of all formulas

Prover	Size	Positive	Mix	Total Time
lprove	13	0.979	0.261	1.24
lprove	14	4.551	5.564	10.116
lprove	15	30.014	5.568	35.583
lprove	16	3053.202	168.074	3221.277
bprove	13	0.943	0.203	1.147
bprove	14	4.461	4.294	8.755
bprove	15	32.206	4.306	36.513
bprove	16	3484.203	129.91	3614.114
dprove	13	5.299	0.798	6.098
dprove	14	23.161	13.514	36.675
dprove	15	107.264	13.645	120.909
dprove	16	1270.586	240.301	1510.887

slower

Prover	Size	Positive	Mix	Total Time
sprove	13	1.757	0.173	1.931
sprove	14	8.037	2.966	11.003
sprove	15	38.266	2.941	41.208
sprove	16	188.317	54.802	243.12
hprove	13	1.007	0.111	1.119
hprove	14	4.413	1.818	6.231
hprove	15	20.09	1.836	21.927
hprove	**16**	**90.595**	**30.713**	**121.308**

faster

Figure: Performance of provers on exhaustive tests

$\Rightarrow$ the Nested Horn Clause form is faster and scalable as size grows

Conclusions and future work

- cross-testing opportunities between:
  - type inference algorithms for lambda terms
  - theorem provers for propositional intuitionistic logic
- our lightweight provers:
  - more likely than provers using complex heuristics, to be turned into parallel implementations
  - provers working on nested Horn clauses outperform those working directly on implicational formulas
  - cover full intuitionistic propositional logic (done, see future paper)
- 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