Shaving with Occam's Razor: Deriving Minimalist Theorem Provers for Minimal Logic

Paul Tarau

University of North Texas

RCRA'2018

Research supported by NSF grant **1423324**

Outline

1. The implicational fragment of propositional intuitionistic logic
2. Proof systems for intuitionistic implicational propositional logic
3. An executable specification
4. Deriving our *lean* theorem provers
5. The testing framework
6. Conclusions and future work

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:

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

and the modus ponens inference rule:

- $MP: \quad A, \ A \rightarrow B \ \vdash \ B.$
- 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**

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

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

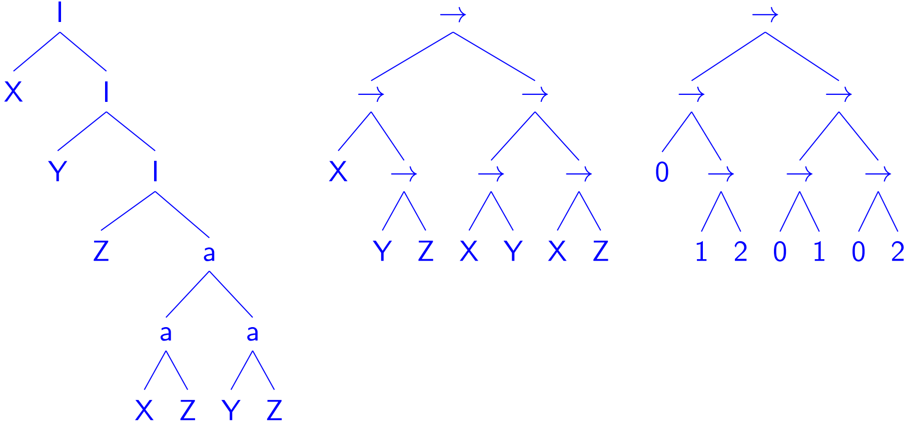
An executable specification

Notations and assumptions

- we use **Prolog** as our meta-language
- code (now grown to above *4000 lines*, covering full propositional logic) at **`https://github.com/ptarau/TypesAndProofs`**
- 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`" 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):- %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]).
```

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

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).
```

Steps for inferring **S** from its type

```
?- s_(S),sprove(S,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:
  - $B_1 \to B_2 \to \ldots \to B_n \to H$ 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).
```

- 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 $x \to \mathit{false}$
- we add the special handling of `false` as the first clause of the predicate `ljb/2`, corresponding to rule

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

```
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

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 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)

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)

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?