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

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. Proving a theorem in intuitionistic propositional logic, with *implication* as its only primitive (also called minimal logic), is known as one of the simplest to state PSPACE-complete problem. At the same time, via the Curry-Howard isomorphism, it is instrumental to finding lambda terms that may inhabit a given type.
However, as hundreds of papers witness it, all starting with Gentzen's **LJ** calculus, conceptual simplicity has not come in this case with comparable computational counterparts. Implementing such theorem provers faces challenges related not only to soundness and completeness but also to termination and scalability problems.
Can a simple solution, in the tradition of "lean theorem provers" be uncovered that matches the conceptual simplicity of the problem, while being able to handle also large randomly generated formulas?
In search for an answer, starting from Dyckhoff's **LJT calculus**, we derive a sequence of minimalist theorem provers using declarative program transformations steps, while highlighting connections, via the Curry-Howard isomorphism, to type inference mechanisms for the simply typed lambda calculus.
We chose Prolog as our meta-language. Being derived from essentially the same formalisms as those we are covering reduces the semantic gap and results in surprisingly concise and efficient declarative implementations. Our implementation is available at: `https://github.com/ptarau/TypesAndProofs`.

Keywords: Curry-Howard isomorphism, propositional implicational intuitionistic logic, type inference and type inhabitation, simply typed lambda terms, theorem proving, declarative algorithms, logic programming, combinatorial search algorithms.

1 Introduction

The implicational fragment of propositional intuitionistic 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:
$MP:\quad A,\ A \to B \vdash B.$

In its simplest form, the Curry-Howard isomorphism [1, 2] connects the implicational fragment of propositional intuitionistic logic (called here *minimal logic*) and types in the *simply typed lambda calculus*. A low polynomial type inference algorithm associates a type (when it exists) to a lambda term. Harder (PSPACE-complete, see [3]) algorithms associate inhabitants to a given type expression with the resulting lambda term (typically in normal form) serving as a witness for the existence of a proof for the corresponding tautology in minimal logic. As a consequence, a theorem prover for minimal logic can also be seen as a tool for program synthesis, as implemented by code extraction algorithms in proof assistants like Coq [4].

Besides the syntactically identical axioms for the combinators S and K and the axioms of the logic, an intuitive reason for this isomorphism between types and formulas is that one can see propositions as set of proofs on which functions corresponding to implications $A \to B$ act as transformers of proofs of A into proofs of B.

Our interest in theorem provers for this minimalist logic fragment has been triggered by its relation via the Curry-Howard isomorphism, to the inverse problem corresponding to inferring types for lambda terms, *type inhabitation*, with direct applications to the generation of simply typed lambda terms that meet a given specification, more efficiently than trying out all possible terms of increasing sizes for a match [5].

At the same time, while generating large lambda terms, usable (e.g., to test scalability of compilers for functional languages), is becoming increasingly difficulty with size as their asymptotic density in the set of closed lambda terms has been shown to converge to 0. As a consequence, even our best generators [6] based on Boltzmann samplers are limited to lambda terms in normal form of about size 60-70, given the very large number of retries needed to filter out untypable terms.

As it the case with SAT-problems, solvable quite efficiently in practice despite being NP-complete, made it appealing to see if the PSPACE-completeness of intuitionistic propositional calculus actually precludes its use at finding types and inhabitants of comparable sizes as those obtained from Boltzmann samplers.

Given the large number of variations coming from trying out different provers we have also focused on setting up an extensive combinatorial and random testing framework to ensure correctness as well as evaluate scalability and performance of our provers.

The paper is organized as follows. Section 2 overviews sequent calculi for implicational propositional intuitionistic logic. Section 3 describes a direct encoding of the LJT calculus as Prolog program. Section 4 describes successive derivation steps leading to simpler and/or faster programs, including embedded Horn clause representations and adaptations to support classical logic via Glivenko's double-negation translation. Section 5 describes our testing framework and section 6 provides performance evaluation. Section 7 overviews related work and section 8 concludes the paper.

2 Proof systems for minimal logic

Initially, Hilbert-style axioms were considered for intuitionistic logic. While simple and directly mapped to SKI-combinators via the Curry-Howard isomorphism, their usability

for automation is very limited. In fact, their inadequacy for formalizing even "hand-written" mathematics was the main trigger of Gentzen's work on natural deduction and sequent calculus, inspired by the needs for formal reasoning in the foundation of mathematics [7].

Thus we start with Gentzen's own calculus for intuitionistic logic, simplified here to only cover the purely implicational fragment, given that our focus is on theorem provers working on formulas that correspond types of simply-typable lambda terms.

2.1 Gentzen's LJ calculus, restricted to the implicational fragment of propositional intuitionistic logic

We assume familiarity with basic sequent calculus [7] notation. Gentzen's original LJ calculus [7] (with the equivalent notation of [8]) uses the following rules.

$$LJ_1: \quad \frac{}{A,\Gamma \vdash A}$$

$$LJ_2: \quad \frac{A,\Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$LJ_3: \quad \frac{A \to B,\Gamma \vdash A \qquad B,\Gamma \vdash G}{A \to B,\Gamma \vdash G}$$

As one can easily see, when trying a goal-driven implementation that uses the rules in upward direction, the unchanged premises on left side of rule LJ_3 would cannot ensure termination as nothing prevents A and G from trading places during the inference process.

2.2 Dyckhoff's LJT calculus, restricted to the implicational fragment of propositional intuitionistic logic

Motivated by problems related to loop avoidance in implementing Gentzen's **LJ** calculus, Roy Dyckhoff [8] splits rule LJ_3 into LJT_3 and LJT_4.

$$LJT_1: \quad \frac{}{A,\Gamma \vdash A}$$

$$LJT_2: \quad \frac{A,\Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$LJT_3: \quad \frac{B,A,\Gamma \vdash G}{A \to B,A,\Gamma \vdash G}$$

$$LJT_4: \quad \frac{D{\rightarrow}B,\Gamma \vdash C{\rightarrow}D \quad B,\Gamma{\rightarrow}G}{(C{\rightarrow}D){\rightarrow}B,\Gamma \vdash G}$$

This avoids the need for loop checking to ensure termination. The rules work with the context Γ being a multiset, but it has been shown later [9] that Γ can be a set, with duplication in contexts eliminated.

For supporting negation, one also needs to add LJT_5 that deals with the special term $false$. Then negation of A is defined as $A \rightarrow false$.

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

Interestingly, as it is not unusual with logic formalisms, the same calculus has been discovered independently in the 50's by Vorob'ev and in the 90's by Hudelmaier.

3 An executable specification: Dyckhoff's LJT calculus, literally

Roy Dyckhoff has implemented the **LJT** calculus as a Prolog program. We have ported it to SWI-Prolog as a reference implementation (see `https://github.com/ptarau/TypesAndProofs/blob/master/third_party/dyckhoff_orig.pro`). However, it is a fairly large (420 lines) program, partly because it covers the full set of intuitionistic connectives and partly because of the complex heuristics that it implements.

This brings up the question if, in the tradition of "lean theorem provers", we can build one directly from the LJT calculus, in a goal oriented style, by reading the rules from conclusions to premises.

Thus, we start with a simple, almost literal translation of rules $LJT_1 \ldots LJT_4$ to Prolog with values in the environment Γ denoted by the variable `Vs`.

```
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_3
  select( ((C->D)->B),Vs1,Vs2),
  ljt((C->D), [(D->B)|Vs2]),
  !,
  ljt(G,[B|Vs2]).
```

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

Note the use of `select/3` to extract a term from the environment (a nondeterministic step) and termination, via a multiset ordering based measure [8]. An example of use is:

```
?- lprove(a->b->a).
true.

?- lprove((a->b)->a).
false.
```

Note also that integers can be used instead of atoms, flexibility that we will use as needed.

Besides the correctness of the bf LJT rule set (as proved in [8]), given that the prover has past our tests, it looks like being already quite close to our interest in a ”lean” prover for minimal logic. However, given the extensive test set (see section 5) that we have developed, it is not hard to get tempted in getting it a bit simpler and faster, knowing that the smallest error will be instantly caught.

4 Simpler, faster = better?

We start with transformations that keep the underlying implicational formula unchanged.

4.1 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).
```

This results in our tests (see section 5 for details) in a 51% speed improvement for formulas with 14 internal nodes.

4.2 Extracting the proof terms

Extracting the proof terms (lambda terms having the formulas we prove as types) is achieved by decorating in the code with application nodes `a/2`, lambda nodes `l/2` (with first argument a logic variable) and leaf nodes (with logic variables, same as the identically named ones in the first argument of the corresponding l/2 nodes).

The simplicity of the predicate `bprove/1` and the fact that this is essentially the inverse of a type inference algorithm (e.g., the one in [10]) help with figuring put how the decoration mechanism works.

```
sprove(T):-sprove(T,_).

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

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

Thus lambda nodes decorate *implication introductions* and application nodes decorate *modus ponens* reductions in the corresponding calculus. Note that the two clauses of `ljs_imp` provide the target node T that, when seen from the type inference side, results from cancelling the source type S and the application type $S \to T$.

Calling `sprove/2` on the formulas corresponding to the types of the S, K and I combinators, we obtain:

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

4.3 From multiset to set contexts

Replacing the multiset context with sets eliminates repeated computations, The larger the terms, the more likely this is to be useful. It combines simplicity of `bprove` and duplicate avoidance via the `add_new/3` predicate.

```
pprove(T):-ljp(T,[]),!.

ljp(A,Vs):-memberchk(A,Vs),!.
ljp((A->B),Vs1):-!,add_new(A,Vs1,Vs2),ljp(B,Vs2).
ljp(G,Vs1):- % atomic(G),
  select((A->B),Vs1,Vs2),
  ljp_imp(A,B,Vs2),
  !,
  add_new(B,Vs2,Vs3),
  ljp(G,Vs3).

ljp_imp(A,_,Vs):-atomic(A),!,memberchk(A,Vs).
ljp_imp((C->D),B,Vs1):-
   add_new((D->B),Vs1,Vs2),
   ljp((C->D),Vs2).

add_new(X,Xs,Ys):-memberchk(X,Xs),!,Ys=Xs.
add_new(X,Xs,[X|Xs]).
```

The performance advantages of this transformation are likely to be significant only as the terms are getting larger.

4.4 Implicational formulas as embedded Horn Clauses

Given the equivalence between: $B_1 \rightarrow B_2 \ldots B_n \rightarrow H$ and (in Prolog notation) H :- $B_1, B_2, \ldots, B_n$, where we chose H is to be the atomic formula ending a chain of implications, we can recursively transform an implicational formula into one built form embedded clauses, as follows.

```
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).
```

Note also that the transformation is reversible and that lists (instead of Prolog's conjunction chains) are used to collect the elements of the body of a clause.

```
?- 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]).
```

This suggests 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 in Vs1
  select((B:-As),Vs1,Vs2),
  select(A,As,Bs),
  ljh_imp(A,B,Vs2),
  !,
  trimmed((B:-Bs),NewB),
  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((A:-[]),A).
trimmed((A:-[B|Bs]),A:-[B|Bs]).
```

Note that we have also added a second `select/3` call to the third clause of `ljx`, to give `ljh_imp` more chances to succeed and commit. Basically, the *embedded Horn clause form of implicational logic 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*. In fact, it might be worth formalizing and studying its properties directly, as a new calculus.

The transformation brings to `hprove/1` an extra 12% performance gain over `bprove/1` on terms of size 14 and a 76% gain on terms of size 15.

4.5 A lifting to classical logic, via Glivenko's transformation

The simplest way to turn a propositional intuitionistic theorem prover into a classical one is to use *Glivenko's translation that prefixes a formula with its double negation*. By adding the atom $false$, to the language of the formulas and a rewriting of the negation of x into $x \rightarrow false$, we obtain, after adding the special handling of `false` as the first line of `ljk/2`:

```
:- op(425,  fy,  ~ ). % negation

gprove(T0):-dneg(T0,T),kprove(T).

kprove(T0):-expand_neg(T0,T),ljk(T,[]),!.

ljk(_,Vs):-memberchk(false,Vs),!.
ljk(A,Vs):-memberchk(A,Vs),!.
ljk((A->B),Vs):-!,ljk(B,[A|Vs]).
ljk(G,Vs1):-
  select((A->B),Vs1,Vs2),
  ljk_imp(A,B,Vs2),
  !,
  ljk(G,[B|Vs2]).

ljk_imp((C->D),B,Vs):-!,ljk((C->D),[(D->B)|Vs]).
ljk_imp(A,_,Vs):-memberchk(A,Vs).
```

```
expand_neg(A,R):-atomic(A),!,R=A.
expand_neg(~A,R):-!,expand_neg(A,B),R=(B->false).
expand_neg((A->B),(X->Y)):-expand_neg(A,X),expand_neg(B,Y).
```

Note that the predicate `kprove/1` simply extends implicational propositional calculus, as implemented by `bprove/1`, with the negation operator `~/1`, while the predicate `gprove/1` prefixes a classical formula with double negation.

5 The testing framework

Correctness can be checked by identifying false positives or false negatives.

While classical tautologies are easily tested (at small scale against truth tables, at medium scale with classical propositional provers and at larger scale with a SAT solver), intuitionistic provers require a more creative approach.

As a first bootstrapping step, assuming that no "gold standard" prover is available one can look at the other side of the Curry-Howard isomorphism, and rely on generators for (typable) lambda terms and generators of formulas for implicational logic expressions, with results being checked against a trusted type inference algorithm.

As a next step, a trusted prover can be used as a gold standard to test both for false positives and negatives.

5.1 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.

We refer to [10] for a detailed description of efficient algorithms generating pairs of simply typed lambda terms in normal form together with their principal types. The variant of the code we use here is at: `https://github.com/ptarau/TypesAndProofs/blob/master/allTypedNFs.pro`

5.2 Finding false positives by generating all well-formed 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. We refer to [10] for a detailed description of the algorithms.

The code describing the all-tree and set partition generation as well as their integration as a type expression generator is at:
`https://github.com/ptarau/TypesAndProofs/blob/master/allPartitions.pro`.

We tested the predicate `lprove/1` as well as all other provers derived from it for false negatives against simple types of terms up to size 15 (with size defined as 2 for

applications, 1 for lambdas and 0 for variables) and for false positives against all type expressions up to size 7 (with size defined as the number of internal nodes).

5.3 Testing against a trusted reference implementation

Assuming we trust an existing reference implementation (e.g., after it passes our generator-based tests), it makes sense to use it as a "gold standard". In this case, we can identify both false positives and negatives directly, as follows:

```
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
  ).
```

When specializing to a generator for all well-formed implication expressions, and using Dyckhoff's dprove/1 predicate as a gold standard, we have:

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

To test the tester, we design a prover that randomly succeeds or fails.

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

We can now test lprove/1 and badprove/1 as follows:

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

More complex implicit correctness tests can be designed, by comparing the behavior of a prover that handles `false`, with Glivenko's double negation transformations that turns an intuitionistic propositional prover into a classical prover, working on classical formulas containing implication and negation operators.

After defining:

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

We can run it against the provers `gprove/1` and `kprove/1`, using Melvin Fitting's classical tautology prover [11] `tautology/1` as a gold standard.

```
?- 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 .
...
```

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

6 Performance and scalability testing

Once passing correctness tests, our provers need to be tested against large random terms. The mechanism is similar to the use of all-term generators.

6.1 Random simply-typed terms, with Boltzmann samplers

We generate random simply-typed normal forms, using a Boltzmann sampler along the lines of that described in [6]. 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),nv(XT).
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.
```

Interestingly, partly due to the fact that there's some variation in the size of the terms Boltzmann sampler generate and more to the fact that the distribution of types favors (as seen in the second example) the simple tautologies where an atom identical to the last one is contained in the implication chain leading to it [12, 13], if we want to use these for scalability tests, additional filtering mechanisms need to be used to statically reject type expressions that are large but easy to prove as intuitionistic tautologies.

6.2 Random implicational formulas

The generation of random implicational formulas is more intricate. Our code combines an implementation of Rémy's algorithm [14], along the lines of Knuth's algorithm **R** in [15] for the *generation of random binary* trees at `https://github.com/ptarau/TypesAndProofs/blob/master/RemyR.pro`. with code to generate *random set partitions* at `https://github.com/ptarau/TypesAndProofs/blob/master/ranPartition.pro`.

We refer to [16] for a declarative implementation of Rémy's algorithm in Prolog with code adapted for this paper at: `https://github.com/ptarau/TypesAndProofs/blob/master/RemyP.pro`.

As automatic Boltzmann sampler generation is limited to fixed numbers of equivalence classes from which a CF- grammar can be given, we build our the random set partition generator that groups logical variables in leaf position into equivalence classes by using an urn-algorithm [17].

Once a random binary tree of size N is generated with the `->/2` constructor labeling internal nodes, the $N+1$ leaves of the tree are decorated with logic variables. As variables sharing a name define equivalence classes on the set of variables, each choice of them corresponds to a set partition of the $N+1$ nodes.

The combined generator, that generates in a few seconds terms of size 1000, works as follows:

```
?- 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,_)). % much faster now, thanks to tabling
% 107,163 inferences,0.040 CPU in 0.044 seconds (92% CPU, 2659329 Lips)
```

Note that we use Prolog's tabling (a form of automated dynamic programming) to avoid costly recomputation of the (very large) Sterling numbers.

6.3 Testing with large random terms

Testing with for false positives and false negatives for random terms proceeds in a similar manner 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
```

In fact, 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. Interestingly, on the side of random simply typed terms, limitations come from their vanishing density, while on the other side they come from the known PSPACE-complete complexity of the proof procedures.

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

Our benchmarking code is at:
`https://github.com/ptarau/TypesAndProofs/blob/master/benchmarks.pro`.

The following table compares several provers on exhaustive "all-terms" benchmarks, derived from our correctness test. First, we run them on the types inferred on all lambda terms of a given size. Next we run them on all implicational formulas of a given size (set to be about half of the former, as the number of these grows much faster).

Prover	Input size	Time on Positive Examples	Time on Mix of Examples	Total Time
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	0.95	0.21	1.16
bprove	14	4.53	4.46	8.99
bprove	15	32.83	4.46	37.3
sprove	13	1.39	0.3	1.69
sprove	14	6.63	6.32	12.95
sprove	15	49.07	6.2	55.28
pprove	13	1.42	0.27	1.69
pprove	14	6.45	5.29	11.75
pprove	15	30.31	5.3	35.62
hprove	13	0.93	0.26	**1.2**
hprove	14	4.13	5.18	**9.32**
hprove	15	19.1	5.18	**24.28**
dprove	13	2.18	0.35	2.53
dprove	14	10.96	6.25	17.21
dprove	15	1100.72	5.76	**1106.49**

Fig. 1. Performance of provers on exhaustive tests

Note that the size of the implicational formulas in the case of the "Mix of examples" is half of the size of the lambda terms whose type are used in the case of "Positive examples"

The embedded Horn clauses based `hprove/1` turns out to be a clear winner. Most likely, this comes from concentrating its non-deterministic choices in the two `select/3` calls. These replace with a faster "shallow backtracking" loop the deep backtracking the other provers might occur to cover the same search space.

In fact, `hprove/1` seems to also scale well on larger formulas, following closely the increase in the numbers of test formulas (with "pos" marking formulas generated

by typable lambda terms of sizes 16,17 and18; ”neg” marking implicational formulas of sizes 8, 8 and 9).

```
?- hbm(16,hprove).
[prog=hprove, size=16, pos=88.26, neg=105.37,total=193.64]

?- hbm(17,hprove).
[prog=hprove, size=17, pos=433.95, neg=109.58,total=543.54]

?- hbm(18,hprove).
[prog=hprove,size=18, pos=2122.84, neg=2413.12,total=4535.96]
```

Testing exhaustively on small formulas, while an accurate indicator for average speed, might not favor provers using more complex heuristics or extensive preprocessing. But if that happens, one would expect it to result in a constant factor ratio, rather than a fast increasing gap as it happens, for instance between Dyckhoff's original `dprove` and our best prover `hprove`.

7 Related work

The related work derived from Gentzen's **LJ** calculus is in the hundreds if not in the thousands of papers and books. Space constraints limit our discussion to the most closely related papers, directly focusing on algorithms for implicational intuitionistic propositional logic, which, as decision procedures, ensure termination without a loop-checking mechanism.

Among them the closest are [8, 9] that we have used as starting points for deriving our provers. We have chosen to implement the **LJT** calculus directly rather than deriving our programs from Roy Dyckhoff's Prolog code.

Similar calculi, key ideas of which made it into the Coq proof assistant's code, are described in [18] and [19].

On the other side of the Curry-Howard isomorphism [20], described in full detail in [21] finds and/or counts inhabitants of simple types in long normal form.

8 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. Among them, the embedded Horn clause prover might be worth formalizing as a calculus and subject to deeper theoretical analysis. Given that it shares its main data structure with Prolog, it seems interesting to attempt for it partial evaluation or even compilation to Prolog via a source-to-source transformation.

Our renewed interest in finding lightweight implementations of these classic theoretically hard (PSPACE-complete) combinatorial search problems, is also motivated on the possibility of parallel implementations using multi-core and GPU algorithms.

We plan future work in formalizing the embedded Horn-clause prover in sequent-calculus and explore compilation techniques and parallel algorithms for it.

A generalization to embedded Horn clauses with universally quantified variables seems also promising to explore, with either grounding techinques as used by SAT and ASPm solvers or via compilation to Prolog.

Acknowledgement

This research has been supported by NSF grant 1423324.

References

1. Howard, W.: The Formulae-as-types Notion of Construction. In Seldin, J., Hindley, J., eds.: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, London (1980) 479–490
2. Wadler, P.: Propositions as types. Commun. ACM **58** (2015) 75–84
3. Statman, R.: Intuitionistic Propositional Logic is Polynomial-Space Complete. Theor. Comput. Sci. **9** (1979) 67–72
4. The Coq development team: The Coq proof assistant reference manual. (2018) Version 8.8.0.
5. Tarau, P.: On Type-directed Generation of Lambda Terms. In De Vos, M., Eiter, T., Lierler, Y., Toni, F., eds.: 31st International Conference on Logic Programming (ICLP 2015), Technical Communications, Cork, Ireland, CEUR (September 2015) available online at http://ceur-ws.org/Vol-1433/.
6. Bendkowski, M., Grygiel, K., Tarau, P.: Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers. TPLP **18**(1) (2018) 97–119
7. Szabo, M.E.: The Collected Papers of Gerhard Gentzen. Philosophy of Science **39**(1) (1972)
8. Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. Journal of Symbolic Logic **57**(3) (1992) 795807
9. 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
10. Tarau, P.: A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms. In Hermenegildo, M.V., Lopez-Garcia, P., eds.: Logic-Based Program Synthesis and Transformation: 26th International Symposium, LOPSTR 2016, Edinburgh, UK, Revised Selected Papers, Springer LNCS, volume 10184 (September 2017) 240–255 , Best paper award.
11. Fitting, M.: leanTAP Revisited. Journal of Logic and Computation **8**(1) (1998) 33–47
12. Genitrini, A., Kozik, J., Zaionc, M.: Intuitionistic vs. Classical Tautologies, Quantitative Comparison. In Miculan, M., Scagnetto, I., Honsell, F., eds.: Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers. Volume 4941 of Lecture Notes in Computer Science., Springer (2007) 100–109
13. Kostrzycka, Z., Zaionc, M.: Asymptotic densities in logic and type theory. Studia Logica **88**(3) (2008) 385–403
14. Rémy, J.L.: Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications **19**(2) (1985) 179–195
15. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming). Addison-Wesley Professional (2006)

16. Tarau, P.: Declarative Algorithms for Generation, Counting and Random Sampling of Term Algebras. In: Proceedings of SAC'18, ACM Symposium on Applied Computing, PL track, Pau, France, ACM (April 2018)
17. Stam, A.: Generation of a random partition of a finite set by an urn model. Journal of Combinatorial Theory, Series A **35**(2) (1983) 231 – 240
18. 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
19. Coquand, C.: From semantics to rules: A machine assisted analysis. In: Selected Papers from the 7th Workshop on Computer Science Logic. CSL '93, London, UK, UK, Springer-Verlag (1994) 91–105
20. Ben-Yelles, C.B.: Type assignment in the lambda-calculus: Syntax and semantics. PhD thesis, University College of Swansea (1979)
21. Hindley, J.R.: Basic Simple Type Theory. Cambridge University Press, New York, NY, USA (1997)