On Lean Theorem Provers for the Implicational Fragment of Propositional Intuitionistic Logic and the Curry-Howard Isomorphism

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
Denton, Texas 76203-5017
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, logic programming, combinatorial search, parallel algorithms.

ACM Reference format:
Paul Tarau. 2018. On Lean Theorem Provers for the Implicational Fragment of Propositional Intuitionistic Logic and the Curry-Howard Isomorphism . In *Proceedings of , , ,* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The implicational fragment of propositional intuitionistic logic can be defined by two axiom schemes:
$K: \ A \to (B \to A)$
$S: \ (A \to (B \to C)) \to ((A \to B) \to (A \to C))$
and the modus ponens inference rule:
$MP: \ A, A \to B \vdash B.$

In its simplest form, the Curry-Howard isomorphism [15, 31] 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 [25]) 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 [20].

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 sets 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 [27].

At the same time, generating large lambda terms, is usable e.g., to test correctness and scalability of compilers for functional languages [6, 22] and proof assistants. But this is becoming increasingly difficult with size, as the asymptotic density of typable terms in the set of closed lambda terms has been shown to converge to 0 [19]. As a consequence, even our best generators [2] 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 SAT-problems are solvable quite efficiently in practice, despite being NP-complete, one might want to see if this extends to the typical PSPACE-complete problem of finding proofs for intuitionistic propositional calculus formulas. If so, an alternate method would

emerge for finding types and inhabitants of comparable sizes as those obtained from Boltzmann samplers [2].

Given the possibly subtle semantic variations between our provers, we have also focused on setting up an extensive combinatorial and random testing framework to ensure correctness, as well as to evaluate their scalability and performance.

The paper's novel contributions include:

- a series of fast, lightweight theorem provers for implicational intuitionistic logic
- several fast and scalable provers using an embedded Horn-clause equivalent for implicational formulas
- a lambda term proof extractor, usable also as a type inhabitation algorithm
- a lean classical propositional tautology prover using Glivenko's double negation translation
- a combinatorial testing framework combining all-term and random-term generators from the two sides of the Curry-Howard isomorphism
- a new random implicational formula generator combining Rémy's algorithm for random generation of binary trees with an urn-based random set partition generator
- parallel algorithms for random generation of large test data
- parallelization of embedded Horn clause-based provers by applying random permutations to clause bodies, statically , before starting the proof process
- fully replicable research claims supported by an open-source code base at `github.com` (tested with `SWI-Prolog 7.7.12`)

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 a 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 Introduces parallel algorithms for both provers and their test data generators. Section 8 overviews related work and section 9 concludes the paper.

2 PROOF SYSTEMS FOR MINIMAL LOGIC

Initially, like for other fields of mathematics and logic, 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 need for formal reasoning in the foundation of mathematics [26].

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 to types of simply-typed lambda terms.

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

We assume familiarity with basic sequent calculus notation. Gentzen's original LJ calculus [26] (with the equivalent notation of [4]) uses the following rules.

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

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

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

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 not ensure termination as nothing prevents A and G from repeatedly 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 [4] splits rule LJ_3 into LJT_3 and LJT_4.

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

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

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

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

This avoids the need for loop checking to ensure termination as one can identify a multiset ordering-based size definition that decreases after each step [4]. The rules work with the context Γ being a multiset, but it has been shown later [5] 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 $A \to false$.

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

Interestingly, as it is not unusual with logic formalisms, the same calculus has been (re)discovered independently in the 50's by Vorob'ev and in the 80's-90's by Hudelmaier [16, 17].

2.3 Notations and assumptions

As we will use **Prolog** as our meta-language, our notations will be derived as much as possible from its syntax (including token types and operator definitions). Thus variables will be denoted with uppercase letters and, as programmer's conventions final "s" letters indicate a plurality of items (e.g., when referring to the content of Γ contexts).

As the code base of this paper has grown at his point to above *2000 lines* (see **https://github.com/ptarau/TypesAndProofs**, our *literate programming* style adopted in the last 10 years was not a realistic option. However, we have tried to bring the most significant code snippets (mostly the actual theorem provers' code) into the paper. This also implies that this time we have to assume that the reader is familiar with basic Prolog programming, including, besides the pure Horn clause subset, well-known built-in predicates like `memberchk/2` and `select/3`, elements of higher order programming (e.g., `call/N`), and occasional use of CUT and `if-then-else` constructs.

Lambda terms are built using the function symbols **a/2**=application, **l/2**=lambda binders with a variable as its first argument and an expression as second and *logic variables* representing the leaf variables bound by a lambda.

Type expressions (also seen as implicational formulas) are built as binary trees with the function symbol "`->/2`" and *logic variables at their leaves*.

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 [4]. 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 **LJT** rule set (as proved in [4]), 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 DERIVING A FEW *LEAN* THEOREM PROVERS

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 [28]), help with figuring out how the decoration mechanism works.

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

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):-
  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: $B_1 \to B_2 \ldots B_n \to H$ 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((B:-[]),R):-!,R=B.
trimmed(BBs,BBs).
```

This time, the 3-rd clause works as a reducer of the assumed hypotheses. It removes from the context a clause `B:-As` and it removes from its body a formula `A`, to be passed to `ljh_imp`, with the remaining context. Should `A` be atomic, we succeed if and only if it is already in the context. Otherwise, we closely mimic rule LJT_4 by trying to

prove `A = D:-Cs`, after extending the context with the assumption `B:-[D]`. Note that in both cases the context gets smaller, as `As` does not contain the `A` anymore. Moreover, should the body `Bs` end up empty, the clause is downgraded to its atomic head.

Note that by having a second `select/3` call in the third clause of `ljh`, will give `ljh_imp` more chances to succeed and commit. Thus, 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.

To take advantage of Prolog's possibly optimized list processing, we can represent our Horn clauses as lists having their head as a first element.

This gives:

```
fprove(T0):-toListHorn(T0,T),ljf(T,[]),!.

ljf(A,Vs):-memberchk(A,Vs),!.
ljf([B|As],Vs1):-!,append(As,Vs1,Vs2),ljf(B,Vs2).
ljf(G,Vs1):- % atomic(G), G not on Vs
  select([B|As],Vs1,Vs2),
  select(A,As,Bs),
  ljf_imp(A,B,Vs2), % A element of the body of B
  !,
  ftrimmed([B|Bs],NewB),
  ljf(G,[NewB|Vs2]).

ljf_imp(A,_B,Vs):-atomic(A),!,memberchk(A,Vs).
ljf_imp([D|Cs],B,Vs):- ljf([D|Cs],[[B,D]|Vs]).
```

It relies into a modified form of `trimmed`.

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

It also relies on a modified transformer from implicational formulas:

```
toListHorn((A->B),[H|Bs]):-!,toListHorns((A->B),Bs,H).
toListHorn(H,H).

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

This uniform list representation is also syntactically somewhat less cluttered:

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

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

We refer to section 6 for an explanation of why this transformation almost doubles the performance compared to `hprove`, as seen, for instance in the following benchmark:

```
?- hbm(14,hprove).
[prog=hprove,size=14,pos=4.1,neg=5.08,total=9.19]
```

```
?- hbm(14,fprove).
[prog=fprove,size=14,pos=3.93,neg=0.46,total=4.4]
```

where the difference comes from the faster rejection of non-theorems (marked with label "neg").

4.5 Hudelmaier's $O\left(n\ log\left(n\right)\right)$-space variant, with fresh variables

While none of our tests, including on large random terms, has indicated resource overflows, we have also implemented a variant of the **LJT** calculus due to [17]. It is derived from `pprove/1` and it works as follows.

```
nprove(T):-ljn(T,[],100,_),!.

ljn(A,Vs)-->{memberchk(A,Vs)},!.
ljn((A->B),Vs1)-->!,{add_new(A,Vs1,Vs2)},ljn(B,Vs2).
ljn(G,Vs1)--> % atomic(G),
  {select((A->B),Vs1,Vs2)},
  ljn_imp(A,B,Vs2),
  !,
  {add_new(B,Vs2,Vs3)},
  ljn(G,Vs3).

ljn_imp(A,_,Vs)-->{atomic(A),!,memberchk(A,Vs)}.
ljn_imp((C->D),B,Vs1)-->newvar(P),
   { add_all([
       C,
       (D->P),
       (P->B)
     ],
     Vs1,Vs2)
   },
   ljn(P,Vs2).
```

The key difference is the introduction of the new variable `P` that ensures the $O\left(n\ log\left(n\right)\right)$-space bound, as shown in [17].

To encapsulate the change of state induced by creation of new variables, we use prolog's DCGs with increments provided by `newvar/3`.

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

While about twice as slow as `pprove` from which it is derived, this prover can be useful if predictable space bounds need to be enforced, for instance in the case one uses fixed memory arrays in a GPU-based implementation.

4.6 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 clause of the predicate `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. At the same time, one can use generators of formulas for implicational logic expressions, with results being checked first 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. In fact, this means that all implicational formulas having proofs shorter than a given number are all covered, but possibly 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.

We refer to [28] 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 from [28] that we use here is at: https://github.com/ptarau/TypesAndProofs/blob/master/allTypedNFs.pro

5.2 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. We refer to [28] for a detailed description of the algorithms.

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.

We have 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 that 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, 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).
```

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) ;
...
```

A more interesting case is when a prover is only guilty of false positives. For instance, let's naively implement the intuition that a goal is provable w.r.t. a context `Vs` if all its premises are provable, with implication introduction assuming premises and success achieved when the environment is reduced to empty.

```
badSolve(A):-badSolve(A,[]).

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).
```

As the following test shows, while no tautology is missed, the false positives are properly caught.

```
?- gold_test(6,badSolve,T,wrong_failure).
false.

?- gold_test(6,badSolve,T,wrong_success).
T =  (0->0->0->0->0->0->1) ;
T =  (0->1->0->0->0->0->2) ;
T =  (0->0->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 transformation 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 [7] `tautology/1` as a gold standard. We have place a variant of it, restricted to implicational logic, at: https://github.com/ptarau/TypesAndProofs/blob/master/third_party/fitting.pro.

```
?- 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 [2]. 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.
```

Interestingly, partly due to the fact that there's some variation in the size of the terms that Boltzmann samplers 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 [11, 19], 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.

6.2 Random implicational formulas

The generation of random implicational formulas is more intricate. Our code combines an implementation of Rémy's algorithm [23], along the lines of Knuth's algorithm **R** in [18] 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 [29] 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 for which a CF- grammar can be given, we build our the random set partition generator that groups variables in leaf position into equivalence classes by using an urn-algorithm [24]. 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 variables denoted by successive integers starting from 0. 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. For instance, a set partition of the leaves `{0,1,2,3}` like `{{0},{1,2},{3}}` will be correspond to the

variable leaf decorations

$$0,1,1,2$$

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] .
```

Note that the list of labels it generates can be directly used to decorate the random 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] .
```

The combined generator, that produces 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,_)). % fast, 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 in the code at https://github.com/ptarau/TypesAndProofs/blob/master/ranPartition.pro.

6.3 Testing with large random terms

Testing 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 of the Curry-Howard isomorphism they come from the PSPACE-complete complexity of the proof procedures.

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

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

Time is measured in seconds. The table in Fig. 1 compares several provers on exhaustive "all-terms" benchmarks, derived from our correctness tests. 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	Size	Positive	Mix	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
nprove	13	2.85	0.38	3.23
nprove	14	13.13	7.32	20.46
nprove	15	63.36	7.31	70.67
hprove	13	0.93	0.26	**1.2**
hprove	14	4.13	5.18	**9.32**
hprove	15	19.1	5.18	**24.28**
fprove	13	0.91	0.03	**0.94**
fprove	14	3.95	0.52	**4.48**
fprove	15	18.2	0.41	**18.61**
dprove	13	2.18	0.35	2.53
dprove	14	10.96	6.25	17.21
dprove	15	1100.72	5.76	**1106.49**

Figure 1: Performance of provers on exhaustive tests

Note that the size of the implicational formulas in the case of the *Mix* of positive and negative examples is half of the size of the lambda terms whose types are used in the case of *Positive* examples, as the former have a much faster growth rate.

The embedded Horn clauses-based `hprove/1 and fprove/1` turn out to be clear winners. Most likely, this comes from concentrating their 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 (see the table in Fig. 2), `hprove/1 and fprove/1` seem to also scale well on larger formulas, following closely the increase in the numbers of test formulas (with *Positive* labeling the column of formulas generated by typable lambda terms of sizes 16,17 and 18; *Mix* labeling the column of implicational formulas of sizes 8, 8 and 9).

The constant but significant advantage of `fprove` over `hprove`, coming from faster rejection of negative examples is likely to originate in list-related Prolog VM optimizations.

Testing exhaustively on small formulas, while an accurate indicator of 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 quickly

Prover	Size	Positive	Mix	Total Time
hprove	16	88.26	105.37	**193.64**
hprove	17	433.95	109.58	**543.54**
hprove	18	2122.84	2413.12	**4535.96**
fprove	16	86.3	7.95	**94.26**
fprove	17	418.32	4.94	**423.26**
fprove	18	2049.92	128.33	**2178.25**

Figure 2: Performance of fastest provers on larger tests

increasing gap, as it happens, for instance, between Dyckhoff's original dprove and our best provers hprove and fprove.

7 ONE MORE THING: A LOOK AT PARALLEL ALGORITHMS FOR PROVERS AND TESTERS

Opportunities for parallel execution abound both in the case of generators and the case of provers.

7.1 Finding the first solution in parallel, with backtracking generators

We shortly discuss here how one can customize a multi-threading package like the one of SWI-Prolog [32] to support backtracking-intensive code.

This is needed as our generators, that, to work space proportional to the size of the generated terms, iterate over billions of alternatives via backtracking and submitting a list of goals known in advance to a predicate like `first_solution/3` (see http://www.swi-prolog.org/pldoc/doc_for?object=thread%3Afirst_solution/3 is unrealistic.

The predicate `nondet_first(Sol,Exec,ExecGen)` combines the work of a generator `ExecGen` and an executor `Exec` that works on alternative candidates and returns the first that succeeds with an answer `Sol`.

```
nondet_first(Sol,Exec,ExecGen):-
        thread_count(ThreadCnt),
        nondet_first_with(ThreadCnt,Sol,Exec,ExecGen).
```

It starts by creating the optimal number of threads (assuming 2/3 of the total number on the machine being available and actually mapped to hardware hyperthreads).

```
thread_count(ThreadCnt):-
  prolog_flag(cpu_count,MaxThreads),
  ThreadCnt is max(2,ceiling((2/3)*MaxThreads)).
```

Then, using message-queues, it sends to each executor a task provided, on backtracking, by the generator.

```
nondet_first_with(ThreadCnt,Sol,Exec,ExecGen):-
        WorkOpt=[],
        SolsOpt=[],
  message_queue_create(Master,WorkOpt),
  message_queue_create(Sols,SolsOpt),
  findall(Id,
    (
      between(1,ThreadCnt,_),
      thread_create(nondet_worker_with(Sols, Master),Id,[])
    ),
  Ids),
  forall(
    ExecGen,              % send as much work as generated
    thread_send_message(Master,Sol-Exec)
  ),
  thread_get_message(Sols,Sol), % receive first solution
  forall( % stop all other threads
    member(Id,Ids),
    catch(thread_signal(Id, abort), _, true)
  ),
  % free resources
  message_queue_destroy(Master),
  message_queue_destroy(Sols).
```

A worker thread receives, as soon as it is ready, a (possibly computationally expensive) goal `G` and runs the executor on it. On success it sends out `Sols`, triggering the master thread to stop all the others.

```
nondet_worker_with(Sols, Master):-
  repeat,
    thread_get_message(Master,Goal),
    Goal=X-G,
    G,
    !,
    thread_send_message(Sols,X),
  fail.
```

7.2 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).
```

While this often generates larger tautologies then those obtained as types of lambda terms using Boltzmann samples, one can also notice the rather short "proof sizes" corresponding to the associated lambda terms. The large number of parallel attempts (in our tests, running on an iMacPro with 18 cores/36 threads) will unavoidably favor random implicational formulas having short proofs over those with longer proofs. So while this method can arguably win over tautologies generated with Boltzmann samplers, these tend to be easier to solve.

7.3 Parallel generation of lambda terms and their types

We can design a parallel version of the random normal form generators, quite easily as searching is a largely independent task.

However, to make our random runs replicable, when dealing with multiple threads, we first generate a set of random seeds that we will use to initialize the (thread-local) random generators.

```
ranseeds(L,Xs):-
  findall(X,
      (between(1,L,_),X is random(2^32)),
  Xs).
```

Then, we customize the program in such a way that, if a specific global random number is given instead of the atom `'random'`, the multi-threaded random generation is fully replicable/deterministic.

```
parRanTypedTNF(Seed,TSize,N,K,X:T,Size):-
  set_random(seed(Seed)),
  MaxSteps=1000000,
  Max is truncate(N*(11/10)),
  Min is truncate(N*(9/10)),
  between(1,K,_),
  parRanTypableNF(Max,Min,TSize,MaxSteps,X,T,Size,_Steps).
```

The actual work is performed by `parRanTypableNF/8` which simply tries as many sequential generators as the number of available threads, using the SWI-Prolog library predicate `first_solution/3`, that returns an answer as soon as the first successful thread terminates.

```
parRanTypableNF(Max,Min,TSize,MaxSteps,X,T,Size,Steps):-
  G=tryRanTypableNF(Max,Min,TSize,MaxSteps,X,T,Size,Steps),
  thread_count(L),
  ranseeds(L,Xs),
  length(Gs,L),
  maplist(add_seed(G),Xs,Gs),
  first_solution(G,Gs,[on_fail(continue)]).

add_seed(G,Seed,(set_random(seed(Seed)),G)).
```

For instance

```
?- 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.
```

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, on an 18 core iMacPro. The lack of exact size determination is a feature of the Boltzmann sampler on which the underlying sequential random simply typed normal form generator [2] is based.

7.4 Parallelizing the provers

Given the small granularity of the search component of our provers, our initial attempts did not indicate performance gains, mostly because in SWI-Prolog, 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.

Therefore, we need to be more creative with designing effective parallel algorithms for our provers and use the fact that that *provability of an implicational formula in an embedded Horn clause form is independent of the ordering of the atoms in the body of the clauses*. Thus, we can statically generate enough variants of those, each likely to influence search order to keep busy a fairly large number of threads, the larger the term, the better.

The predicate `toRandomHorn` converts an implicational formula to an embedded Horn clause equivalent by shuffling the bodies of the embedded Horn clauses.

```
toRandomHorn((A->B),(H:-Xs)):-!,
   toRandomHorns((A->B),Bs,H),
   trimHorn(H,Bs,Ts),
   sort(Ts,Cs),
   random_permutation(Cs,Xs).
toRandomHorn(H,H).

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

Moreover, we can statically reduce the search space, by eliminating duplicates with `sort/2` and by compressing trivially true formulas like $H:-B_1,B_2,...H...B_n$ to the shorter true formula $H:-H$.

```
trimHorn(A,Bs,R):-memberchk(A,Bs),!,R=[A].
trimHorn(_,Bs,Bs).
```

The transformation works as follows:

```
?- toRandomHorn((0->1->(2->0->(1->2->3)->4->2)->5->2),R).
R =  (2:-[0, 1, 5,  (2:-[2])]).

?- toRandomHorn((0->1->(2->0->(1->2->3)->4->2)->5->2),R).
R =  (2:-[1,  (2:-[2]), 5, 0]).
```

The parallelization of a prover (using an embedded Horn clause representation) follows easily as:

```
parProveHorn(T):-parProveHorn(hprove,T).

parProveHorn(P,T):-
 thread_count(K),
 Sol=YesNo,
 ExecGen=ranHornPermuted(K,T,RT),
 Exec=proveYesNo(P,RT,YesNo),
 nondet_first_with(K,Sol,Exec,ExecGen),
 Sol=true.

proveYesNo(P,T,YesNo):-
  call(P,T)->YesNo=true
; YesNo=false.
```

Note the use in this case of the nondeterministic first solution finder `nondet_first_with/4` that launches exactly `K` shuffled variants based on availability of `K` threads. An example of use is:

```
?- parProveHorn((0->1->(2->0->(1->2->3)->0->4->2)->5->2)).
false.

?- parProveHorn((0->1->2->(2->0->(1->2->3)->0->4->2)->5->2)).
true.
```

Parallel execution is especially useful on very large, unprovable formulas, where the parallel algorithm often succeeds quickly when the corresponding sequential provers take a very long time.

```
?- ihard(Hard),time(parProveHorn(hprove,Hard)).
% 25,442 inferences, 0.012 CPU in 0.684 seconds ...
false.

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

In this case, the implicational formula of size 100 (see code at: https://github.com/ptarau/TypesAndProofs/blob/master/tester.pro) is instantly solved with the parallelized `hprove/1` while it takes more than 2000 seconds when `hprove/1` is executed sequentially.

8 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 [4, 5], 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. At he same time, as in Roy Dyckhoff's original prover, we have benefitted from the elegant, loop-avoiding rewriting step also present in Hudelmaier's work [16, 17].

Similar calculi, key ideas of which made it into the Coq proof assistant's code, are described in [12] and [3].

On the other side of the Curry-Howard isomorphism, [1] described in full detail in [13], finds and/or counts inhabitants of simple types in long normal form. But 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.

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 [9, 10], followed by a series of Lambda-Prolog and linear logic-related books and papers, e.g., [14, 21]. The similarity to the propositional subsets of N-Prolog [9] and λ-Prolog [21] comes from their close connection to intuitionistic logic, although neither derive implementations from a pure **LJ**-based calculus or have termination properties implemented along the lines the **LJT** calculus. In [30] backtrackable linear and intuitionistic assumptions that mimic the implication introduction rule are used, but they do not involve arbitrarily deep embedded implicational formulas.

Overviews of closely related calculi, using the implicational subset of propositional intuitionistic logic are [5, 8].

9 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.

As a more general pattern, as complexity classes are based on worst-case complexity, parallelizing lean algorithms for problems with tractable the average cases can outperform complex heuristics.

As a novel element, statically generating independent instances that reshuffle commutative operations (e.g., order in the body of a Horn clause) was shown to be beneficial in solving very large problems. While provers working on embedded Horn clauses also outperformed sequentially those working directly on implicational formulas, this static optimization for parallel processing made them able to instantly solve hard problems on which their sequential variants can take thousands of seconds.

Our lightweight implementations of these classic theoretically hard (PSPACE-complete) combinatorial search problems, are likely to also enable parallel implementations using multi-core and GPU algorithms.

Among our provers, the embedded Horn clause provers might be worth formalizing as a calculus and subject to deeper theoretical analysis. Given that they share their main data structures with Prolog, it seems interesting to attempt their partial evaluation or even compilation to Prolog via a source-to-source transformation.

We plan future work in formalizing the embedded Horn-clause prover in sequent-calculus and exploring compilation techniques and new parallel algorithms for it. A generalization to embedded Horn clauses with conjunctions and universally quantified variables seems also promising to explore, especially with grounding techniques as used by SAT and ASP solvers, or via compilation to Prolog.

ACKNOWLEDGEMENT

This research has been supported by NSF grant 1423324.

REFERENCES

[1] Choukri-Bey Ben-Yelles. 1979. *Type assignment in the lambda-calculus: Syntax and semantics*. Ph.D. Dissertation. University College of Swansea.

[2] Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. 2018. Random generation of closed simply typed λ-terms: A synergy between logic programming and Boltzmann samplers. *TPLP* 18, 1 (2018), 97–119. https://doi.org/10.1017/S147106841700045X

[3] Catarina Coquand. 1994. From Semantics to Rules: A Machine Assisted Analysis. In *Selected Papers from the 7th Workshop on Computer Science Logic (CSL '93)*. Springer-Verlag, London, UK, UK, 91–105.

[4] Roy Dyckhoff. 1992. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic* 57, 3 (1992), 795âĂŞ807. https://doi.org/10.2307/2275431

[5] Roy Dyckhoff. 2016. Intuitionistic Decision Procedures Since Gentzen. In *Advances in Proof Theory*, Reinhard Kahle, Thomas Strahm, and Thomas Studer (Eds.). Springer International Publishing, Cham, 245–267.

[6] Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 383–405.

[7] Melvin Fitting. 1998. leanTAP Revisited. *Journal of Logic and Computation* 8, 1 (1998), 33–47.

[8] Dov Gabbay and Nicola Olivetti. 2002. Goal-oriented deductions. In *Handbook of Philosophical Logic*. Springer, 199–285.

[9] Dov M. Gabbay. 1985. N-PROLOG: An extension of PROLOG with hypothetical implication. II. Logical foundations, and negation as failure. *The Journal of Logic Programming* 2, 4 (1985), 251–283.

[10] Dov M Gabbay and Uwe Reyle. 1984. N-Prolog: An extension of Prolog with hypothetical implications. I. *The Journal of Logic Programming* 1, 4 (1984), 319–355.

[11] Antoine Genitrini, Jakub Kozik, and Marek Zaionc. 2007. Intuitionistic vs. Classical Tautologies, Quantitative Comparison. In *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers (Lecture Notes in Computer Science)*, Marino Miculan, Ivan Scagnetto, and Furio Honsell (Eds.), Vol. 4941. Springer, 100–109. https://doi.org/10.1007/978-3-540-68103-8_7

[12] Hugo Herbelin. 1995. 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)*. Springer-Verlag, London, UK, UK, 61–75.

[13] J. Roger Hindley. 1997. *Basic Simple Type Theory*. Cambridge University Press, New York, NY, USA.

[14] Joshua S. Hodas and Dale Miller. 1994. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Journal of Information and Computation* 110, 2 (May 1994), 327–365.

[15] W.A. Howard. 1980. The Formulae-as-types Notion of Construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.P. Seldin and J.R. Hindley (Eds.). Academic Press, London, 479–490.

[16] Hudelmaier. 1988. *A PROLOG Program for Intuitionistic Logic*. Universität Tübingen. https://books.google.com/books?id=__2KPgAACAAJ

[17] J org Hudelmaier. 1993. An O(n log n)-Space Decision Procedure for Intuitionistic Propositional Logic. *Journal of Logic and Computation* 3, 1 (1993), 63–75.

[18] Donald E. Knuth. 2006. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional.

[19] Zofia Kostrzycka and Marek Zaionc. 2008. Asymptotic Densities in Logic and Type Theory. *Studia Logica* 88, 3 (2008), 385–403. https://doi.org/10.1007/s11225-008-9110-0

[20] The Coq development team. 2018. *The Coq proof assistant reference manual*. http://coq.inria.fr Version 8.8.0.

[21] Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA.

[22] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST'11)*. ACM, New York, NY, USA, 91–97.

[23] Jean-Luc Rémy. 1985. 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.

[24] A.J Stam. 1983. Generation of a random partition of a finite set by an urn model. *Journal of Combinatorial Theory, Series A* 35, 2 (1983), 231 – 240.

[25] Richard Statman. 1979. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theor. Comput. Sci.* 9 (1979), 67–72. https://doi.org/10.1016/0304-3975(79)90006-9

[26] M. E. Szabo. 1972. The Collected Papers of Gerhard Gentzen. *Philosophy of Science* 39, 1 (1972).

[27] Paul Tarau. 2015. On Type-directed Generation of Lambda Terms. In *31st International Conference on Logic Programming (ICLP 2015), Technical Communications*, Marina De Vos, Thomas Eiter, Yuliya Lierler, and Francesca Toni (Eds.). CEUR, Cork, Ireland. available online at http://ceur-ws.org/Vol-1433/.

[28] Paul Tarau. 2017. A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms. In *Logic-Based Program Synthesis and Transformation: 26th International Symposium, LOPSTR 2016, Edinburgh, UK, Revised Selected Papers*, Manuel V Hermenegildo and Pedro Lopez-Garcia (Eds.). Springer LNCS, volume 10184, 240–255. https://doi.org/10.1007/978-3-319-63139-4_14 , Best paper award.

[29] Paul Tarau. 2018. Declarative Algorithms for Generation, Counting and Random Sampling of Term Algebras. In *Proceedings of SAC'18, ACM Symposium on Applied Computing, PL track*. ACM, Pau, France.

[30] Paul Tarau, Veronica Dahl, and Andrew Fall. 1996. Backtrackable State with Linear Affine Implication and Assumption Grammars. In *Concurrency and Parallelism, Programming, Networking, and Security (Lecture Notes in Computer Science 1179)*, Joxan Jaffar and Roland H.C. Yap (Eds.). Springer, Berlin Heidelberg, 53–64.

[31] Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58 (2015), 75–84.

[32] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjorn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12 (1 2012), 67–96. Issue Special Issue 1-2. https://doi.org/10.1017/S1471068411000494