A Combinatorial Testing Framework for Intuitionistic Propositional Theorem Provers

Paul Tarau

University of North Texas

PADL’2019

Research supported by NSF grant 1423324
Overview

- In search for an efficient but minimalist theorem prover, on the two sides of the Curry-Howard isomorphism, we design a combinatorial testing framework using types inferred for lambda terms as well as all-term and random term generators.

- We choose Prolog as our meta-language. Being derived from essentially the same formalisms as those we are covering, it reduces the semantic gap and results in surprisingly concise and efficient declarative implementations.

- Our implementation is available at: https://github.com/ptarau/TypesAndProofs.
Provers for implicational fragment of propositional intuitionistic logic: test-driven derivation steps from proof systems to executable code
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**: \( A \rightarrow (B \rightarrow A) \)
- **S**: \( (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)) \)

and the modus ponens inference rule:

- **MP**: \( A, A \rightarrow B \vdash B \).

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 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

⇒

- a lambda term (typically in normal form) can serve as a witness for the existence of a proof for the corresponding tautology in the implicational fragment of propositional intuitionistic logic
Gentzen’s $\text{LJ}$ calculus, reduced to the implicational fragment of intuitionistic propositional logic

- $\text{LJ}_1 : \quad A, \Gamma \vdash A$
- $\text{LJ}_2 : \quad \Gamma \vdash A \rightarrow B \quad \Rightarrow \quad \Gamma \vdash A \rightarrow B$
- $\text{LJ}_3 : \quad A \rightarrow B, \Gamma \vdash A \quad B, \Gamma \vdash G \quad \Rightarrow \quad A \rightarrow B, \Gamma \vdash G$

- rules, if implemented directly are subject to looping
- several variants use loop-checking, by recording the sequents used
Roy Dyckhoff’s **LJT** calculus (implicational fragment)

- replace **LJ**₃ with **LJT**₃ and **LJT**₄
- termination proven using multiset orderings
- no need for loop checking
- efficient and simple

**LJT**₁ : \[ A, \Gamma \vdash A \]

**LJT**₂ : \[ A, \Gamma \vdash B \quad \Gamma \vdash A \rightarrow B \]

**LJT**₃ : \[ B, A, \Gamma \vdash G \quad A \rightarrow B, A, \Gamma \vdash G \quad [ A \text{ atomic } ] \]

**LJT**₄ : \[ D \rightarrow B, \Gamma \vdash C \rightarrow D \quad B, \Gamma \rightarrow G \quad (C \rightarrow D) \rightarrow B, \Gamma \vdash G \]

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

**LJT**₅ : \[ \text{false}, \Gamma \vdash G \]
Prolog as a meta-language: 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”, atoms or integers as their leaves

our code is at: https://github.com/ptarau/TypesAndProofs
the $S$ combinator (left) and its type (right, with integers as leaves):

```
[  
```

```
[  
  [  [ 0, 1, 2, 0, 1, 0, 2 ] ] ]
```
The first step: from Sequent Calculus to Prolog

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

\( \Rightarrow \)

- we start with a simple, almost literal translation of rules \( \mathbf{LJT}_1 \ldots \mathbf{LJT}_4 \) to Prolog
- note: values in the environment \( \Gamma \) denoted by the variables \( V_s, V_{s1}, V_{s2} \ldots \).
Roy Dyckhoff’s LJT calculus, literally

\[ \text{lprove}(T) : \text{-} \text{ljt}(T, [])!, !. \]

\[ \text{ljt}(A, Vs) : \text{-} \text{memberchk}(A, Vs), !. \quad \text{\% LJT\_1} \]

\[ \text{ljt}((A\rightarrow B), Vs) : \text{-} !, \text{ljt}(B, [A|Vs]). \quad \text{\% LJT\_2} \]

\[ \text{ljt}(G, Vs1) : \text{-} \text{\%atomic}(G), \]
\[ \text{\% LJT\_3} \]
\[ \text{select}((A\rightarrow B), Vs1, Vs2), \]
\[ \text{atomic}(A), \]
\[ \text{memberchk}(A, Vs2), \]
\[ !, \]
\[ \text{ljt}(G, [B|Vs2]). \]

\[ \text{ljt}(G, Vs1) : \text{-} \text{\% LJT\_4} \]
\[ \text{select}(((C\rightarrow D)\rightarrow B), Vs1, Vs2), \]
\[ \text{ljt}((C\rightarrow D), [(D\rightarrow B)|Vs2]), \]
\[ !, \]
\[ \text{ljt}(G, [B|Vs2]). \]
**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).
```
**sprove**: extracting the proof terms

```prolog
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(_,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

\[
\text{?- sprove (((0->1->2)->(0->1)->0->2),X).}
\]
\[
X = l(A, l(B, l(C, a(a(A, C), a(B, C)))))). \quad \% \text{S}
\]

\[
\text{?- sprove ((0->1->0),X).}
\]
\[
X = l(A, l(B, A)). \quad \% \text{K}
\]

\[
\text{?- sprove ((0->0),X).}
\]
\[
X = l(A, A). \quad \% \text{I}
\]
Steps for inferring the lambda term for \textbf{S} from its type

?- s_(SType),sprove(SType,STerm).

% [Term:Type,...]  -->  Goal to prove

[]  -->  A: (0\rightarrow1\rightarrow2)\rightarrow(0\rightarrow1)\rightarrow0\rightarrow2
[A: (0\rightarrow1\rightarrow2)]  -->  B: (0\rightarrow1)\rightarrow0\rightarrow2
[A: (0\rightarrow1), B: (0\rightarrow1\rightarrow2)]  -->  C: (0\rightarrow2)
[A: 0, B: (0\rightarrow1), C: (0\rightarrow1\rightarrow2)]  -->  D:2
[a(A,B):1, B:0, C: (0\rightarrow1\rightarrow2)]  -->  D:2
[a(A,B): (1\rightarrow2), a(C,B):1, B:0]  -->  D:2
[a(a(A,B),a(C,B)):2, a(C,B):1, B:0]  -->  D:2

SType = ((0\rightarrow1\rightarrow2)\rightarrow(0\rightarrow1)\rightarrow0\rightarrow2),
STerm = l(A, l(B, l(C, a(a(A, C), a(B, C))))).
Implicational formulas as nested Horn Clauses

- equivalence between:
  - \( B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_n \rightarrow H \) and
  - \( H :\neg B_1, B_2, \ldots, B_n \) (in Prolog notation)

- \( H \) is the atomic formula ending a chain of implications

- we can recursively transform an implicational formula:

\[
toHorn\((A\rightarrow B), (H:-Bs)) :-!, toHorns\((A\rightarrow B), Bs, H)\).
toHorn\((H,H)\).
\]

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

?- toHorn\(((0\rightarrow1\rightarrow2)\rightarrow(0\rightarrow1)\rightarrow0\rightarrow2), R)\).
R = (2:-[(2:-[0, 1]), (1:-[0]), 0]).

?- toHorn\(((0\rightarrow1\rightarrow2\rightarrow3\rightarrow4)\rightarrow(0\rightarrow1\rightarrow2)\rightarrow0\rightarrow2\rightarrow3), R)\).
R = (3:-[(4:-[0, 1, 2, 3]), (2:-[0, 1]), 0, 2]).

- also, note that the transformation is reversible!
Transforming provers for implicational formulas into equivalent provers working on nested Horn clauses

\[
hprove(T0) :- \text{toHorn}(T0, T), \text{ljh}(T, []), !. \\
\]

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

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

\[
\text{trimmed}((B:-[]), R) :- !, R=B. \\
\text{trimmed}(BBs, BBs). \\
\]
What’s *new* with the nested Horn clause form?

The *nested Horn clause form helps bypassing some intermediate steps, by focusing on the head of the Horn clause, which corresponds to the last atom in a chain of implications*. 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
The combinatorial testing framework
Combinatorial testing, automated

- **testing correctness:**
  - a false positive: it is not a tautology, but the prover proves it
  - a false negative: it is a tautology but the prover fails on it
  - no false positive: a prover is **sound**
  - no false negative: a prover is **complete**
  - soundness and completeness are relative to a "gold standard"!

- **helpers:**
  - intuitionistic tautologies are also classical, so if it is not classical it cannot be intuitionistic
  - crossing the Curry-Howard bridge: types of all lambda terms up to a given size: types of simply typed lambda terms are tautologies for sure

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

- finding false negatives by generating the set of simply typed normal forms of a given size
- finding false positives by generating all implicational formulas/type expressions of a given size
- testing against a trusted reference implementation
- random simply-typed terms, with Boltzmann samplers
- generating random implicational formulas
- scalability tests
Finding false negatives by generating the set of simply typed normal forms of a given size

- a false negative is identified if our prover fails on a type expression known to have an inhabitant
- via the Curry-Howard isomorphism, such terms are the types inferred for lambda terms, generated by increasing sizes
- this means that all implicational formulas having proofs shorter than a given number are covered
- but, small formulas having long proofs might not be reachable with this method that explores the search by the size of the proof rather than the size of the formula to be proven!
Finding false positives by generating all implicational formulas/type expressions of a given size

- a false positive is identified if the prover succeeds finding an inhabitant for a type expression that does not have one.
- we obtain type expressions by generating all binary trees of a given size, extracting their leaf variables and then iterating over the set of their set partitions, while unifying variables belonging to the same partition
- code at: https://github.com/ptarau/TypesAndProofs/blob/master/allPartitions.pro.
- an advantage of exhaustive testing with all formulas of a given size is that it implicitly ensures coverage: no path is missed simply because there are no paths left unexplored
- but, we need an oracle that tells as which formulas should succeed and which should fail!
- ⇒ we need a trusted reference implementation!
Testing against a trusted reference implementation

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

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

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

Paul Tarau (University of North Texas) Combinatorial Testing for Theorem Provers PADL’2019 24 / 29
Random simply-typed terms, with Boltzmann samplers

- once passing correctness tests, our provers need to be tested against large random terms (a scalability test)
- we generate random simply-typed normal forms, using a Boltzmann sampler
  code is at: https://github.com/ptarau/TypesAndProofs/blob/master/ranNormalForms.pro

?- ranTNF(60,XT,TypeSize).
XT = l(l(a(a(0, l(a(a(0, a(0, l(...))), s(s(0)))))), s(s(0)))),
    l(l(a(a(0, a(l(...)), a(..., ...))), l(0))))):
    (A→(((A→A)− ...)→D)→D)→M)→M),
TypeSize = 34.
Random implicational formulas from binary trees and set partitions

- The combined generator, produces in a few seconds terms of size 1000:

\[- \text{ranImpFormula}(20,F).\]
\[F = (((0->((1->2)->1->2->2)->3)->2)->4->(3->3)->(5->2)->6->3)->7->(4->5)->(4->8)->8) .\]

\[- \text{time( ranImpFormula}(1000,\_)).\]
% includes tabling large Stirling numbers
% 37,245,709 inferences, 7.501 CPU in 7.975 seconds (94% CPU, 4965628 Lips)

\[- \text{time( ranImpFormula}(1000,\_)).\] % fast, thanks to tabling
% 107,163 inferences, 0.040 CPU in 0.044 seconds (92% CPU, 2659329 Lips)

- superexponential growth with $N$, Catalan($N$)*Bell($N+1$)
Scalability testing: a quick performance evaluation

- **our benchmarking code is at:** [https://github.com/ptarau/TypesAndProofs/blob/master/bm.pro](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 \( \frac{N}{2} \)
  - for the winner, we also test it on larger formulas up to size 20 and 10
Runtimes on known tautologies and mix of all formulas

<table>
<thead>
<tr>
<th>Prover</th>
<th>Size</th>
<th>Positive</th>
<th>Mix</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lprove</td>
<td>13</td>
<td>0.979</td>
<td>0.261</td>
<td>1.24</td>
</tr>
<tr>
<td>lprove</td>
<td>14</td>
<td>4.551</td>
<td>5.564</td>
<td>10.116</td>
</tr>
<tr>
<td>lprove</td>
<td>15</td>
<td>30.014</td>
<td>5.568</td>
<td>35.583</td>
</tr>
<tr>
<td>lprove</td>
<td>16</td>
<td>3053.202</td>
<td>168.074</td>
<td>3221.277</td>
</tr>
<tr>
<td>bprove</td>
<td>13</td>
<td>0.943</td>
<td>0.203</td>
<td>1.147</td>
</tr>
<tr>
<td>bprove</td>
<td>14</td>
<td>4.461</td>
<td>4.294</td>
<td>8.755</td>
</tr>
<tr>
<td>bprove</td>
<td>15</td>
<td>32.206</td>
<td>4.306</td>
<td>36.513</td>
</tr>
<tr>
<td>bprove</td>
<td>16</td>
<td>3484.203</td>
<td>129.91</td>
<td>3614.114</td>
</tr>
<tr>
<td>dprove</td>
<td>13</td>
<td>5.299</td>
<td>0.798</td>
<td>6.098</td>
</tr>
<tr>
<td>dprove</td>
<td>14</td>
<td>23.161</td>
<td>13.514</td>
<td>36.675</td>
</tr>
<tr>
<td>dprove</td>
<td>15</td>
<td>107.264</td>
<td>13.645</td>
<td>120.909</td>
</tr>
<tr>
<td>dprove</td>
<td>16</td>
<td>1270.586</td>
<td>240.301</td>
<td>1510.887</td>
</tr>
<tr>
<td>sprove</td>
<td>13</td>
<td>1.757</td>
<td>0.173</td>
<td>1.931</td>
</tr>
<tr>
<td>sprove</td>
<td>14</td>
<td>8.037</td>
<td>2.966</td>
<td>11.003</td>
</tr>
<tr>
<td>sprove</td>
<td>15</td>
<td>38.266</td>
<td>2.941</td>
<td>41.208</td>
</tr>
<tr>
<td>sprove</td>
<td>16</td>
<td>188.317</td>
<td>54.802</td>
<td>243.12</td>
</tr>
<tr>
<td>hprove</td>
<td>13</td>
<td>1.007</td>
<td>0.111</td>
<td>1.119</td>
</tr>
<tr>
<td>hprove</td>
<td>14</td>
<td>4.413</td>
<td>1.818</td>
<td>6.231</td>
</tr>
<tr>
<td>hprove</td>
<td>15</td>
<td>20.09</td>
<td>1.836</td>
<td>21.927</td>
</tr>
<tr>
<td>hprove</td>
<td>16</td>
<td>90.595</td>
<td>30.713</td>
<td>121.308</td>
</tr>
</tbody>
</table>

Figure: Performance of provers on exhaustive tests

⇒ the Nested Horn Clause form is faster and scalable as size grows
Conclusions and future work

- our test-driven development framework is built on cross-testing opportunities between type inference algorithms for lambda terms and theorem provers for propositional intuitionistic logic
- our lightweight implementations are more likely than provers using complex heuristics, to be turned into parallel implementations using multi-core and GPU algorithms
- provers working on nested Horn clauses outperformed those working directly on implicational formulas
- future work: formally describing the nested Horn-clause prover in sequent-calculus as well as exploring compilation techniques and new parallel algorithms for it
- a generalization to nested 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