

A History-Based Theorem Prover for Intuitionistic Propositional Logic using Global Caching: IntHistGC System Description

Rajeev Goré, Jimmy Thomson and Jesse Wu

Research School of Computer Science, Australian National University

Abstract. We describe an implementation of a new theorem prover for Intuitionistic Propositional Logic based on a sequent calculus with histories due to Corsi and Tassi. The main novelty of the prover lies in its use of dependency directed backtracking for global caching. We analyse the performance of the prover, and various optimisations, in comparison to current state of the art theorem provers and show that it produces competitive results on many classes of formulae.

1 Preliminaries

IntHistGC is a theorem prover for Intuitionistic Propositional Logic (Int), based on a sound and cut-free complete sequent calculus which uses histories to guarantee termination. The key element behind the prover’s efficiency is the use of global caching to reduce search space. This system description provides an overview of IntHistGC’s proof strategy, implementation, optimisations and an analysis of performance in comparison to the current best provers for Int.

Sequent Calculus. Figure 1 presents the standard multiple conclusioned sequent system for Int using sets of formulae. These rules form the basic elements of the calculus used in our implementation. Note that we assume $\neg\varphi := (\varphi \rightarrow \perp)$.

The $\rightarrow L$ rule may result in infinite looping since the principal formula $\varphi \rightarrow \psi$ must be copied into the left premise for completeness. Various approaches are known to ensure termination: explicit loop-checking; Vorob’ev’s method [1]; and the use of complex histories to track loops [2, 3].

We take a different approach, by replacing the basic $\rightarrow L$ and $\rightarrow R$ rules with those shown in Figure 2. The $\rightarrow L$ rule is replaced by a “blocking” version which prevents any further (backward) applications until “unblocked” by a $\rightarrow R$ -first rule. The $\rightarrow R$ -rest rule can only (and must) be applied (upwards) if we have previously applied $\rightarrow R$ -first on the same implication on the same branch.

Rules in this calculus either delete a connective or block an implication. Implications are only unblocked after the use of a $\rightarrow R$ -first rule, which is prevented from being applied to the same formula twice. Hence there cannot be an infinite sequence of blocking and unblocking, so backward proof search in this calculus always terminates. Although this calculus was developed independently, recently we have learnt that it is functionally identical to the sequent calculus IG [4].

Lemma 1. $\top$, $\perp$, $\wedge$, $\vee$, id , $\rightarrow L$ -blocking and $\rightarrow R$ -rest are invertible rules.

$$\begin{array}{c}
\frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta} \top L \quad \frac{}{\Gamma, \perp \vdash \Delta} \perp L \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \perp, \Delta} \perp R \quad \frac{}{\Gamma \vdash \top, \Delta} \top R \\
\\
\frac{}{\Gamma, \varphi \vdash \varphi, \Delta} id \quad \frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta} \wedge L \quad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta} \wedge R \\
\\
\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta} \vee L \quad \frac{\Gamma \vdash \varphi, \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta} \vee R \\
\\
\frac{\Gamma, \varphi \rightarrow \psi \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \rightarrow \psi \vdash \Delta} \rightarrow L \quad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi, \Delta} \rightarrow R
\end{array}$$

Fig. 1: Basic sequent calculus rules.

$$\begin{array}{c}
\frac{\Gamma, \varphi \rightarrow_b \psi \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \rightarrow \psi \vdash \Delta} \rightarrow L\text{-blocking} \quad \frac{\Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \rightarrow \psi, \Delta} \rightarrow R\text{-rest} \\
\\
\frac{\mathbf{p}, \varphi_1 \rightarrow \psi_1, \dots, \varphi_n \rightarrow \psi_n, \varphi \vdash \psi}{\mathbf{p}, \varphi_1 \rightarrow_b \psi_1, \dots, \varphi_n \rightarrow_b \psi_n \vdash \varphi \rightarrow \psi, \mathbf{q}, \varphi_{n+1} \rightarrow \psi_{n+1}, \dots, \varphi_{n+m} \rightarrow \psi_{n+m}} \rightarrow R\text{-first}
\end{array}$$

Fig. 2: New rules where $\mathbf{p}$ and $\mathbf{q}$ represent disjoint sets of atomic propositions.

Backward Proof Search Strategy. Our proof strategy is as below. Note that backtracking is required at $\rightarrow R$ -first jumps, as the rule is not semantically invertible:

while some rule is applicable to a leaf sequent **do**
 stop: apply any applicable termination rule (id , $\perp L$, $\top R$) to that leaf
 saturate: else apply any applicable static rule ($\top L$, $\perp R$, $\wedge L$, $\vee R$, $\rightarrow L$ -blocking, $\rightarrow R$ -rest) to that leaf
 step: else apply the transitional rule ($\rightarrow R$ -first) to that leaf

Theorem 1. *Our rules are sound and our strategy is complete with respect to Int [4], and produces $O(n^2)$ depth proofs.*

Related Work. We now describe the main techniques behind three state of the art theorem provers for Int. All use some form of pre-processing: see Section 2.

BDDIntKt checks the validity of a formula φ_0 by constructing a finite closure $cl(\varphi_0)$ and checking if any of the subsets in the closure can lead to a model which falsifies φ_0 [5]. While the closure generally contains exponentially many subsets, by using Binary Decision Diagrams (BDDs) many of these do not necessarily have to be explicitly created.

fCube is a Prolog prover based on a signed tableaux calculus for Int [6]. The prover makes heavy use of a variety of simplification rules, whereby formulae with known sign (as well as atoms of a certain polarity) may be replaced under certain conditions with $\top$ or $\perp$. These rules produce an equivalent but simpler sequent, and can significantly reduce unnecessary branching and backtracking.

Imogen uses formula polarity to restrict applicable inference rules [7]. The prover is based on a focused inverse method, which reduces the search space in a sequent calculus based on the polarity of connectives and atomic propositions. Proof search is applied in a forward manner, unlike our strategy above.

2 Implementation and Optimisations

The sequent rules described in Figures 1 and 2, and the general search strategy of Section 1, were implemented in Ocaml. Search is conducted in a depth-first manner, fully exploring one branch before considering others. While breadth-first search is also possible, a depth-first strategy allows for a natural implementation of global caching [8] as described next. IntHistGC includes a (slower) configuration which produces graphs for both valid and invalid derivations.

Pre-processing - A simple processing function which replaces inputs with equivalent formulae can significantly reduce the complexity of a proof. By imposing a standard ordering on all formulae, one can completely remove any complications due to variable ordering at commutative operators. For example, the formula $(\varphi \wedge \psi) \rightarrow (\psi \wedge \varphi)$ can be rewritten as $(\varphi \wedge \psi) \rightarrow (\varphi \wedge \psi)$ which simplifies to $\top$. IntHistGC also makes use of standard logical identities, such as $\varphi \wedge \top = \top \wedge \varphi = \varphi \wedge \varphi = \varphi$, and removes nested implications using the identity $\varphi_0 \rightarrow (\varphi_1 \rightarrow \varphi_2) = (\varphi_0 \wedge \varphi_1) \rightarrow \varphi_2$ before applying backwards proof search.

Dependency directed backtracking and Caching. The application of sequent rules often leads to proofs which only require a subset of formulae within a sequent. If two valid sequents differ only by formulae which are not necessary for a proof of validity, then a proof for one is sufficient to prove the other. *Thrashing* occurs when branches with the same important formulae are redundantly explored. Backjumping is a technique which avoids this phenomenon, by backtracking to the last point which contains all formulae relevant for a proof [9]. This technique is applied at all branching points, to reduce search space as much as possible.

We extend backjumping by also implementing dependency directed backtracking [9] which stores sets of formulae used to close branches. This is done by placing minimal sequents, like those in backjumping, into a cache. Thereafter, any sequent containing formulae which are a superset of a cached sequent can be proved by applying the same rules. This removes the need to continue search within that branch, and so the prover can return to the latest backtracking point.

In addition to a cache for provable sequents, we also implement a cache for unprovable sequents. In contrast to the minimal sequents constructed for the provable cache, here we include as many formulae as possible; as long as the overall sequent is still not provable. If any sequent is a subset of a previously unproven sequent, then it also cannot be proven - at best, all the same rules will be applied, which we know from previous experience cannot lead to a proof.

Both caches are global and thus search space can potentially be reduced across all branches, by substituting previous proofs. However, caching introduces its own problems. Caching sequents after all rule applications will result in an exponential growth in cache size. To somewhat mitigate unnecessary growth in cache size, cache additions are only considered at $\rightarrow R$ -first jumps and branching points where backjumping has been applied. As each branch terminates without loops, we do not need the full complications of global caching for modal logics [8].

Data Structures. Formulae are represented using (positive) integers. After pre-processing, each formula (including subformulae) is given a unique identifier, and a number of tables are used to map integers to their respective formula, their formula type, as well as the integers corresponding to any subformulae.

We use a set-trie data structure for caching, and implement an adaptation of the algorithms presented in [10]. As formulae are integers, sequents can be represented by an ordered list of integers. A set-trie takes advantage of such an ordering, allowing efficient storage of sequents and set containment searches.

Heuristics. For efficiency, non-branching rules are prioritised over branching rules. Formulae are also expanded in a lazy fashion: only the top-most connective of a formula is ever considered. Thus the prover can avoid some memory issues when expanding bi-implications, interpreted as $\varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

Some branches are inherently easier to prove (or disprove) than others. For example, by changing the order of branch choice in a naïve depth-first search, formulae of the form $(\neg(\neg a_1) \leftrightarrow a_2) \leftrightarrow (a_2 \leftrightarrow a_1)$ can take either a few milliseconds or more than ten minutes to solve - a clear indication of the importance of branch choice. Currently our implementation takes an approach based on formula order as defined in the pre-processing stage.

Rule choice is similarly important, and is considered in implementation. For example, when more than one $\rightarrow$ L-blocking rule is applicable, our prover attempts to pick one which corresponds to an application of modus ponens. Such a choice essentially allows the rule *id* to be immediately applied to one branch.

3 Experimental Results

Performance is evaluated using the Intuitionistic Logic Theorem Proving (ILTP) Library [11], which allows a maximum of 600 seconds for each individual problem instance. We present results in Figure 3 for our prover (IH) with varying optimisations, along with results for `BDDIntKt -autoreorder -nary -assumimp` (BDD), `fCube` (no options) and `Imogen -h optimize`. We also include experiments on an unofficial extended set of the ILTP problem classes, kindly provided by the authors of the library, Jens Otten and Thomas Rath (http://www.iltp.de/download/SYJ2xx-50/SYJ2xx-50.tar.gz) in Figure 4. All tests were conducted on a machine with an Intel i7-3770 CPU @3.40GHz with 8GB memory. The configurations for IntHistGC are as follows:

Configuration	Optimisations involved
naïve	The baseline prover without any optimisations
b	Backjumping
c	Global caching with additions made at $\rightarrow$ R-first jumps
c2	Global caching with additions also at backjumping points

Figure 3 clearly shows that backjumping and caching both improve significantly upon our naïve implementation, allowing the prover to complete the majority of the ILTP benchmark. In fact, global caching by itself, without backjumping,

Class	naïve	IHb	IHc	IHbc	IHbc2	BDD	fCube	Imogen	Out of
LCL	2	2	2	2	2	2	2	2	2
SYN	20	20	20	20	20	20	20	20	20
SYJ10*	12	12	12	12	12	12	12	12	12
SYJ201	2	20	20	20	20	20	20	20	20
SYJ202	3	5	3	5	8	13	9	8	20
SYJ203	10	20	20	20	20	20	20	20	20
SYJ204	20	20	20	20	20	20	20	20	20
SYJ205	10	20	20	20	20	20	20	20	20
SYJ206	5	20	20	20	20	20	20	20	20
SYJ207	2	3	20	20	20	20	20	20	20
SYJ208	10	10	20	20	20	11	20	20	20
SYJ209	9	10	20	20	20	20	20	20	20
SYJ210	20	20	20	20	20	20	20	20	20
SYJ211	3	4	20	20	20	20	20	20	20
SYJ212	20	20	20	20	20	20	20	20	20
Total	148	206	257	259	262	258	263	262	274

Fig. 3: Performance of various versions of IntHistGC on the ILTP benchmark.

manages to solve all problems except that of the valid pigeonhole class SYJ202. Note that IntHistGC’s ‘c’ configuration has no positive effect on this class, as there is only one top-level $\rightarrow$ R-first jump.

While caching clearly has benefits, profiling reveals that the majority of the time is generally spent on cache look-ups. Therefore we implemented a ‘c3’ option which performs cache look-ups every second node, rather than at all branching points. While such a heuristic may result in cache misses, Figures 4 and 6 shows that this reduction in look-up times often outweighs the extra branching involved since such branching is often mitigated by backjumping from a later cache hit.

We remark that the ILTP library is limited to 12 main classes of formulae, half of which are only slight modifications of the other classes. In light of this, we also benchmarked on other families of formulae: Portia, which are encodings of a “real world puzzle”; Nishimura [12], previously used by the authors of fCube; and randomly generated formulae, noting that these tend to be non-valid. Figures 5 and 6 presents timing performance on these classes, in addition to hard instances of the ILTP problems. Particularly noteworthy are the provable Portia examples, which show that both IntHistGC and fCube seem to be susceptible to input ordering. Nevertheless, while using very different techniques to improve speed, fCube and IntHistGC currently seem to be the best provers for Int (Figures 4, 5 and 6). The two trade wins in pigeonhole and Nishimura, although fCube is superior on (typically non-valid) random formulae (Figure 5).

Ideally, a prover which utilises the best of both caching and simplification would perform very well. Unfortunately the two techniques are somewhat conflicting, as simplification introduces new formulae which often result in cache misses. Indeed, we implemented the replacement and basic permanence rules used by fCube [13] but these resulted in worse performance than caching alone.

Class	IHc	IHbc	IHbcc3	IHbc2	BDD	fCube	Imogen	Out of
SYJ201	50	50	50	50	50	50	36	50
SYJ202	3	5	6	8	13	9	8	38
SYJ203	50	50	50	50	50	50	50	50
SYJ204	50	50	50	50	50	50	50	50
SYJ205	50	50	50	50	50	50	50	50
SYJ206	50	50	50	50	50	50	50	50
SYJ207	50	50	50	50	49	50	26	50
SYJ208	38	38	38	38	11	38	37	38
SYJ209	50	50	50	50	50	50	50	50
SYJ210	50	50	50	50	50	50	50	50
SYJ211	50	50	50	50	45	50	50	50
SYJ212	50	50	50	50	50	50	46	50
Total	541	543	544	546	518	547	503	576

Fig. 4: Performance on extended ILTP formulae.

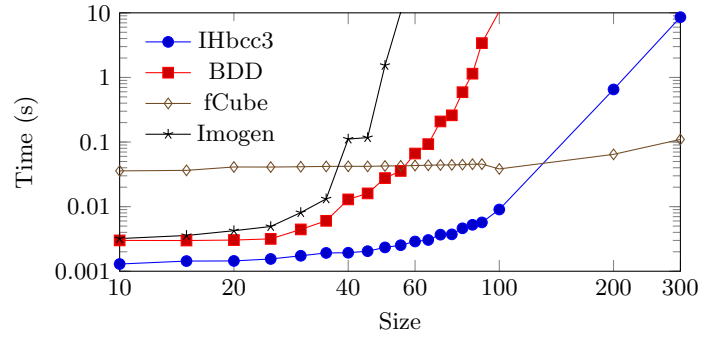


Fig. 5: Average times for random formula of increasing size.

Problem	IHbc	IHbc2	IHbcc3	IHbc2c3	BDD	fCube	Imogen
SYJ201+1.050	5.69	10.13	4.58	6.87	31.794	159.92	—
SYJ202+1.008	—	190.16	—	208.21	0.33	27.86	70.76
SYJ207+1.050	301.23	454.69	161.73	242.08	188.36	17.90	—
SYJ208+1.038	181.58	185.99	113.32	112.70	—	1.76	—
SYJ211+1.050	2.18	1.37	10.15	6.17	—	0.02	0.27
SYJ212+1.050	0.26	0.26	0.22	0.22	50.91	0.22	—
p-portia.100	429.97	—	238.00	—	—	334.02	—
n-portia.100	5.32	295.84	4.52	14.64	—	385.12	—
n-portiaV2.100	443.25	—	239.08	—	—	97.62	—
nishimura.035	0.51	0.51	0.51	0.51	7.05	—	—

Fig. 6: Timing comparisons (all times in seconds, — indicates a timeout)
‘p’ means provable, ‘n’ is non-provable and V2 is a version with reordered formulae

4 Conclusion and Further Work

We have introduced a new prover, which demonstrates the practicality of a history based sequent calculus using global caching. We find that various optimisation techniques make a significant difference in terms of performance. In particular, global caching was extremely effective on the ILTP benchmark. To our knowledge, there are no other ATPs for Int which similarly utilise caching.

Further research into the interaction between simplification and caching techniques merits investigation. The development of a technique which allows caching and simplification to fully complement each other should have a significant impact on prover efficiency. Other than this, it may also be interesting to examine the performance of global caching using a different underlying calculus.

References

- [1] Dyckhoff, R. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.* 57 (1992) 795–807.
- [2] Howe, J.M. Two loop detection mechanisms: A comparison. In: *TABLEAUX*. LNCS 1227. Springer (1997) 188–200.
- [3] Heuerding, A., Seyfried, M., Zimmermann, H. Efficient loop-check for backward proof search in some non-classical propositional logics. In: *TABLEAUX*. LNCS 1071. Springer (1996) 210–225.
- [4] Corsi, G., Tassi, G. Intuitionistic logic freed of all metarules. *J. Symb. Log.* 72 (2007) 1204–1218.
- [5] Goré, R., Thomson, J. BDD-based automated reasoning for propositional bi-intuitionistic tense logics. In: *IJCAR*. LNCS 7364. Springer (2012) 301–315.
- [6] Ferrari, M., Fiorentini, C., Fiorino, G. fCube: An efficient prover for intuitionistic propositional logic. In: *LPAR*. LNCS 6397. Springer (2010) 294–301.
- [7] McLaughlin, S., Pfenning, F. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In: *LPAR*. LNCS 5330. Springer (2008) 174–181.
- [8] Goré, R., Nguyen, L.A. Exptime tableaux for ALC using sound global caching. *J. Autom. Reasoning* 50 (2013) 355–381.
- [9] Hustadt, U., Schmidt, R.A. Simplification and backjumping in modal tableau. In: *TABLEAUX*. LNCS 1397. Springer (1998) 187–201.
- [10] Savnik, I. Index data structure for fast subset and superset queries. In: *Availability, Reliability, and Security in Information Systems and HCI*. LNCS 8127. Springer (2013) 134–148.
- [11] Rath, T., Otten, J., Kreitz, C. The ILTP problem library for intuitionistic logic. *J. Autom. Reasoning* 38 (2007) 261–271.
- [12] Fiorentini, C. All intermediate logics with extra axioms in one variable, except eight, are not strongly ω -complete. *J. Symb. Log.* 65 (2000) 1576–1604.
- [13] Ferrari, M., Fiorentini, C., Fiorino, G. Simplification rules for intuitionistic propositional tableaux. *ACM Trans. Comput. Log.* 13 (2012) 14.