EasyChair Preprint

№ 3926

# lazyCoP 0.1

Michael Rawson and Giles Reger

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

July 22, 2020

# lazyCoP 0.1

Michael Rawson Giles Reger

July 21, 2020

**Abstract**

We describe lazyCoP, a fully-automatic theorem prover for first-order logic with equality. The system implements a connection-tableau calculus with a specific variant of ordered paramodulation inference rules, rather than the usual preprocessing approaches. We explore practical aspects and refinements of this calculus. The system also implements fully-parallel proof search and support for the integration of learned search heuristics.

## 1 Background and Motivation

Systems such as leanCoP [10], nanoCoP [13], or SETHEO [4] which implement the model-elimination/connection calculus [5] have a number of compelling advantages: goal-directed proof search, natural parallelism opportunities [18], adaptation to other logics [10, 12], potential re-use of Prolog technology and resulting compact implementations [10], and increasingly-succesful application of machine learning techniques for heuristic search [3].

The treatment of equality in such calculi is not as succesful, however: the popular *paramodulation* approach [1, 8] (and associated refinements such as superposition) is not complete for the connection calculus if applied naïvely [14]. Various other techniques are used in practice, typically before proof search begins: leanCoP at present simply adds voluminous and potentially-explosive equality axioms to the problem, but many techniques claim to improve on this [9].

We approach this area from attempting to create a practical theorem prover guided by a neural network heuristic. Such networks and their hardware acceleration typically introduce significant latency to proof search [7], but in principle this latency can be hidden by exploiting parallelism at the proof search level [15]. When implementing this new system it seemed appropriate to explore at least one alternative technique for equality reasoning in connection calculi. Paskevich's "lazy paramodulation" system LPCT [14] appears promising in theory and was previously unexplored in practice as far as the authors are aware.

## 2 Proof Calculus

LPCT achieves completeness by postponing unification steps until a later stage: if $P(t_1, t_2, \ldots, t_n)$ must conventionally unify with $P(s_1, s_2, \ldots, s_n)$,

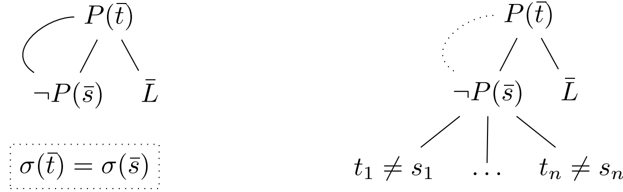


Figure 1: Conventional "strict" and LPCT "lazy" extension steps.

LPCT instead generates new literals $t_i \neq s_i$ to refute before proceeding. In the case where sub-terms do unify, the generated literals may still be unified with an equality reduction rule. However, this approach allows the sub-terms to be further rewritten by paramodulation before unification, preserving completeness. Figure 1 shows the result of an extension step in both the conventional connection calculus and in LPCT.

## 2.1 Classical Refinements

Research on connection tableaux has produced a large number of powerful refinements of the raw calculus [5]. Where possible we adapt these to the LPCT calculus for practical implementation in lazyCoP:

**Start clauses.** Trivially, lazyCoP can still begin with the conjecture if available. Otherwise, all *negative* clause are selected: $t \neq t$ may be refuted in isolation, but $t = t$ cannot.

**Tautology elimination.** This refinement eliminates tableaux containing a clause which has become tautologous through unification. We extend this to include reflexive tautologies $t = s$ with $\sigma(t) = \sigma(s)$.

**Path regularity.** No literal can occur twice on the active path. This is also enforced for equality literals up to symmetry: if $t = s$ is on the active path, both $t = s$ and $s = t$ are forbidden.

**Enforced folding-up.** *Hochklappen* is implemented in lazyCoP.

**Strong regularity.** Extension steps using literals (including equalities) that could be reduced via path/lemmata literals are eliminated.

## 2.2 Further Extensions

We also implement a few other techniques which seem to help in some cases. If $t \neq s$ is on the active path, $t$ may not be equal to $s$ at any point: otherwise, the literal could have been closed immediately. Further, LPCT produces ordering constraints $l \succ r$ on paramodulations: we add the constraint that if $l = r$ is paramodulating onto $s[p] = t$, $s[p] \succ t$. This is a similar approach to the superposition calculus, but not quite identical.

One criticism of LPCT might be that proofs can become significantly longer, especially in the non-equality case, as predicate unifications take place incrementally in argument order. lazyCoP implements both conventional "strict" and LPCT "lazy" versions of all rules. The resulting

duplication can be eliminated somewhat by restricting "lazy" rules to not simulate "strict" rules. It is not clear what effect this has on proof search: changes to calculi rarely bring about a monotonic improvement and the duplication adds significant implementation complexity.

## 2.3 Completeness

LPCT is known to be complete without any refinements of the core calculus. However, the status of LPCT with the described refinements (and by extension lazyCoP) is not known and we do not argue either way. However, empirically it appears that these refinements do not prevent finding proofs and are very useful in practice.

# 3 Implementation

Research into practical implementations of similar systems make significant effort to avoid recomputing results. Our experience was different: by taking a simplistic approach to both algorithms and data structures, the resulting system was simpler and faster. This may be due to implementation incompetence, but perhaps also due to changes in processor and compiler technology. Modern processors are clocked higher with access to more cores and memory than their predecessors, but also rely more heavily on memory caching, branch prediction and instruction-level parallelism.

For example, the *variable trail* is frequently exhorted as a necessity for efficient unification with backtracking [5], but this is a relatively complex structure. Most of the time we simply avoid backtracking and instead rebuild the tableau from scratch, but while making deductions from a parent tableau one-step backtracking to the parent cannot be avoided efficiently. Instead we make a copy of the necessary parent structures, then destructively modify them to create a child. The parent can then be restored with a few block memory-copy operations from the saved structures.

To this end almost all data structures in the theorem prover are stored as contiguous blocks of memory ("arenas") which are resident for the whole proof run. This approach nearly eliminates allocation overhead and significantly improves cache locality. Worker threads have exclusive access to their own blocks, reducing contention. The Rust [2] programming language allows this level of control over data layout while providing high-level features and prohibiting some classes of memory-/thread-safety bugs.

## 3.1 Representations

Representing the syntactic structure of terms, literals and clauses such that all required operations are efficient is challenging. lazyCoP solves this Prolog-style by storing a "term graph" of multiple terms sharing variables in an array: variables are identified only by their position in the array, and compound terms store a symbol and offsets to their arguments relative to their position in the array, shown in Figure 2. Literals and subsequently clauses index into this array.

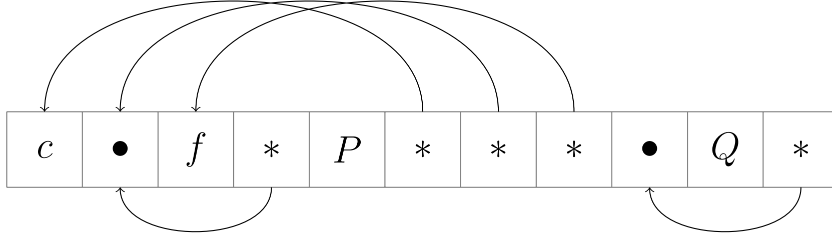


Figure 2: A term graph for $P(c, x, f(x))$ after appending $Q(y)$.

This representation is unwieldy, but allows for fast implementations. Copying clauses up to variable renaming, a bottleneck for some systems, is particularly efficient: appending a term graph to the end of an existing term graph renames all variables so that they are disjoint.

## 3.2 Constraints

At the core of lazyCoP is a constraint-solving engine. There are three types of binary constraint generated as rules are applied to tableaux, either to express unifications or to implement refinements. *Equality* constraints require unifying two terms, *ordering* constraints require that one term be larger than another under a reduction ordering $\succ$, and *disequation* constraints require that two terms not become equal under substitution.

Disequation constraints are placed into *solved form* [5] before checking under substitutions. The reduction ordering is the lexicographic path ordering [8], implemented[1] with the optimised algorithm [6] used in E [17].

## 3.3 Search and Parallelism

Proof search in lazyCoP is not yet particularly advanced, although later we hope to integrate learned heuristic guidance. For simplicity we use the traditional A* heuristic search algorithm to explore tableau space and do not attempt to re-order subgoals: the admissible heuristic is the number of open branches.

However, search is fully parallel: lazyCoP spawns one worker thread per core when starting to search. Each worker has copies of data that must be mutated during search (i.e. tableau syntax, constraints), but shares read-only data such as problem clauses. The only mutable piece of shared state is the priority queue of unexpanded tableaux: this is protected with a fast mutex. Queue operations are typically fast relative to other prover mechanisms, so lazyCoP scales near-linearly with available CPU cores.

[1] with thanks to Stephan Schulz for supplying materials and encouragement

### 3.4 Indexing

Due to the lazy nature of inferences in LPCT, only the top symbol of terms is required to match. Furthermore only the problem clauses need to be indexed, resulting in a relatively small, static set compared to other term-indexing settings [19]. Therefore we take an approach usually known as "top-symbol hashing" in the literature: a lookup table maps symbols (and polarities in the predicate case) to lists of clause/literal positions within the problem. This simple and cache-efficient method works well on a variety of problem domains, achieving perfect filtering for lazy rules.

### 3.5 Clausification

Translation to clause-normal form is handled externally with the Vampire [16] system. Some settings were tweaked for the benefit of lazyCoP: Vampire is not permitted to delete the conjecture during pre-processing, and the threshold for introducing names is reduced. However, the author of the similar leanCoP system reports [10] that a custom definitional transformation can produce much better performance.

## 4 Future Work

Significant work can be done to improve the performance of the system. For example, leanCoP implements a custom clausification routine, a strategy schedule, and restricted backtracking [11], none of which lazyCoP yet implements. Older systems such as SETHEO also have a wealth of literature to explore.

The system has been designed around the idea of heuristic assistance from a coprocessor (i.e. a neural network forward-pass evaluated on a GPU), but this is not implemented in lazyCoP 0.1. Such heuristic guidance could significantly improve search on target problems once trained on similar easier problems.

## 5 Acknowledgements

The first author is indebted to Geoff Sutcliffe and Martin Riener for testing, and to Jens Otten for his cheerful correspondence.

## 6 Conclusion

lazyCoP in this initial version is already a plausible system for automated reasoning tasks. Further work is required to exploit the potential of this approach, not least its *raison d'être*: machine-learned guidance.

# References


[1] A. Degtyarev and A. Voronkov. What you always wanted to know about rigid E-unification. *Journal of Automated Reasoning*, 20(1-2):47–80, 1998.

[2] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[3] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, pages 8822–8833, 2018.

[4] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[5] R. Letz and G. Stenz. Model elimination and connection tableau procedures. In *Handbook of Automated Reasoning*, volume 2. MIT Press, 2001.

[6] B. Löchner. Things to know when implementing LPO. *International Journal on Artificial Intelligence Tools*, 15(01):53–79, 2006.

[7] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

[8] R. Neuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, volume 1. MIT Press, 2001.

[9] B. E. Oliver and J. Otten. Equality preprocessing in connection calculi. In *Practical Aspects of Automated Reasoning*, 2020.

[10] J. Otten. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In *International Joint Conference on Automated Reasoning*, pages 283–291. Springer, 2008.

[11] J. Otten. Restricting backtracking in connection calculi. *AI Communications*, 23(2-3):159–182, 2010.

[12] J. Otten. MleanCoP: A connection prover for first-order modal logic. In *International Joint Conference on Automated Reasoning*, pages 269–276. Springer, 2014.

[13] J. Otten. nanoCoP: A non-clausal connection prover. In *International Joint Conference on Automated Reasoning*, pages 302–312. Springer, 2016.

[14] A. Paskevich. Connection tableaux with lazy paramodulation. *Journal of Automated Reasoning*, 40(2-3):179–194, 2008.

[15] M. Rawson and G. Reger. A neurally-guided, parallel theorem prover. In *International Symposium on Frontiers of Combining Systems*, pages 40–56. Springer, 2019.

[16] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI communications*, 15(2, 3):91–110, 2002.

[17] S. Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

[18] J. Schumann, A. Wolf, and C. Suttner. Parallel theorem provers based on SETHEO. In *Automated Deduction—A Basis for Applications*, pages 261–290. Springer, 1998.

[19] R. Sekar, I. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, volume 2. MIT Press, 2001.