Efficient Low-level Connection Tableaux

Cezary Kaliszyk

Institute of Computer Science
University of Innsbruck
Austria

Abstract. Many tableaux provers that follow Stickel's *Prolog Technology* and *lean* have been relying on the Prolog compiler for an efficient term representation and the implementation of unification. In particular, this is the case for leanCoP, the only tableaux prover that regularly takes part in the CASC, the yearly ATP competition. On the other hand, the most efficient superposition provers are typically written in low-level languages, reckoning that the efficiency factor is significant.

In this paper we discuss low-level representations for first-order tableaux theorem proving and present the Bare Metal Tableaux Prover, a C implementation of the exact calculus used in the leanCoP theorem prover with its cut semantics. The data structures are designed in such a way that the prove function does not need to allocate any memory. The code is less elegant than the Prolog code, albeit concise and readable. We also measure the constant factor that a high-level programming language incurs: the low-level implementation performs 18 times more inferences per second on an average TPTP CNF problem. We also discuss the implementation improvements which could be enabled by complete access to the internal data structures, such as direct manipulation of backtracking points.

1 Introduction

Connection tableaux is a well-studied calculus for automating first-order classical logic proofs. An implementation of this calculus, the leanCoP [10] theorem prover, achieves noteworthy performance while keeping the code compact. Since 2007 leanCoP 2.0 [9] has been regularly taking part in the CASC yearly ATP competition, typically performing average in the first-order theorems category [14,15].

leanCoP is implemented in Prolog and relies on the Prolog engine to implement terms, syntactic equality checking, unification, and backtracking efficiently (a number of Prolog compilers and interpreters are supported). The implementation follows the *lean* approach: clauses are stored in the Prolog database to make use of Prolog's indexing. On the one hand, this allows for elegant and very concise code: the main `prove` function of leanCoP needs only about 20 lines of code. On the other hand, the optimizations possible in the implementation might be limited by what can be realized in an elegant way in Prolog. This is in sharp contrast with the provers that typically win the first-order division of

CASC [15]: They are either entirely implemented in low-level languages, such as C in case of E-Prover [13], C++ in case of Vampire [6], or include an efficient low-level core, such as a SAT-solver used inside iProver [5]. Even if some of the low-level implementations perform worse than leanCoP, the low-level implementations of the best performing provers suggest, that the constant factor implied by the choice of the programming language may be significant.

To evaluate this factor, we reimplemented the core of the leanCoP theorem prover, together with its cut semantics, in C. Starting our experiment, we expected the Prolog compilers to optimize the code very well, consequently we were not sure that a low-level implementation would be faster. Already with a simple implementation, we observed a significant improvement w.r.t. the number of performed inferences per second. This made us experiment with the low-level implementation further: We used a memory-efficient representation of terms and clauses. We added perfect sharing of terms and clauses. We used the Robinson's unification algorithm [12] with simple repetition checking, as is was shown to be most effective for first-order theorem proving in practice [2]. We made sure all functions satisfy the requirements of sibling-call optimization (a restriction of tail-call optimization supported by most C compilers) and made sure that no memory is allocated throughout the core proving process.

The C equivalent of the Prolog `prove` function is much less elegant, however it is significantly more efficient: We have modified the low-level implementation and the Prolog implementation of the leanCoP calculus, to ensure that they create the same matrix for the same CNF problems and confirmed that the two implementations perform precisely the same inferences on the same problems. For connection tableaux proofs, the code produced by an optimizing C compiler can perform 18 times more inferences per second than that produced by a Prolog compiler. The imperative implementation also enables optimizations and modifications to the algorithm that are not easily possible in Prolog.

The rest of the paper is structured as follows: In section 2 we present leanCoP and its calculus. In section 3 we discuss the choices made in the implementation of our Bare Metal Tableaux Prover and present the code of the core loop. In section 4 we evaluate the implementation and compare it with a Prolog implementation on a large subset of TPTP. Finally in section 5 we discuss modifications and optimizations to the algorithm that are enabled by an imperative implementation and conclude.

2 leanCoP and Restricted Backtracking

leanCoP implements a clause connection tableaux calculus [7,10] presented in Fig. 1. The *Reduction* rule connects a literal on the current path with the complement of the literal to solve. The *Extension* rule performs a clausal extension step unifying one of the newly attached literals with the complement of the literal to solve. The basic calculus is additionally extended by a lemma rule, that allows to solve a literal that is identical to a previously solved one.

$$\frac{}{\{\},\ M,\ Path}\ \textit{Axiom}$$

$$\frac{C,\ M,\ \{\}}{M}\ \textit{Start}\quad \text{where } C \in M, C \text{ is positive}$$

$$\frac{C,\ M,\ Path \cup \{L_2\}}{C \cup \{L_1\},\ M,\ Path \cup \{L_2\}}\ \textit{Reduction}\quad \text{where } \sigma(L_1) = \sigma(\overline{L_2})$$

$$\frac{C_2 \setminus \{L_2\},\ M,\ Path \cup \{L_1\} \qquad C,\ M,\ Path}{C \cup \{L_1\},\ M,\ Path}\ \textit{Extension}\quad \text{where } \begin{array}{c}\sigma(L_1) = \sigma(\overline{L_2}),\\ \sigma \text{ is rigid},\\ C_1 \in M, L_2 \in C_2,\\ C_2 \text{ is a copy of } C_1\\ \text{with variables renamed}\end{array}$$

Fig. 1. The clause connection calculus used in leanCoP.

In the following, we will shortly explain the Prolog implementation of the calculus in leanCoP and explain restricted backtracking. The `prove` function, presented in Fig. 2, takes as the first argument the clause to prove. If the clause is empty, the proof succeeds (line 1). Otherwise the clause is split into the first literal `Lit` and the rest of the clause `Cla`. The algorithm first checks for regularity (line 5). Next, one of the three cases needs to be fulfilled: either the literal `Lit` is among the already covered lemmas (line 7), or its complement – `NegLit` – unifies with a literal on the path (line 9), or `NegLit` unifies with one of the literals in the matrix[1] and the rest of the unifying clause is recursively provable (lines 13–16). If any of the above three alternatives is successful, we still need to continue with the rest of the clause, which is done by a recursive call to `prove` (line 19).

An important part of the algorithm implemented by leanCoP is the restriction of the search space by the means of a Prolog cut (line 18). Cut is a Prolog built-in predicate that always succeeds, but cannot be backtracked. The most successful strategy in leanCoP uses cut after the application of lemma, reduction, and extension rules. This restricts the backtracking, allowing for significantly increased number of successfully solved TPTP problems [9]. However, as the strategies that involve cut introduce incompleteness, they are typically used in combination with complete strategies.

3 Low-level Implementation

We have implemented an equivalent of the Prolog `prove` predicate in C including all the necessary prerequisites: CNF literals, clauses, syntactic equality checking, unification, and Prolog backtracking. In this section we discuss these components. The low-level implementation does not include a TPTP problem parser and the preparation of the matrix, as these typically are not costly in

[1] leanCoP stores an association table between toplevel predicates and the rests of the clauses. It is referred to as matrix.

```
1  prove([],_,_,_,_,[]).
2
3  prove([Lit|Cla],Path,PathLim,Lem,Set,Proof) :-
4      Proof=[[[NegLit|Cla1]|Proof1]|Proof2],
5      \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
6      (-NegLit=Lit;-Lit=NegLit) ->
7         ( member(LitL,Lem), Lit==LitL, Cla1=[], Proof1=[]
8           ;
9           member(NegL,Path), unify_with_occurs_check(NegL,NegLit),
10          Cla1=[], Proof1=[]
11          ;
12          lit(NegLit,NegL,Cla1,Grnd1),
13          unify_with_occurs_check(NegL,NegLit),
14          ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
15            \+ pathlim -> assert(pathlim), fail ),
16          prove(Cla1,[Lit|Path],PathLim,Lem,Set,Proof1)
17        ),
18        ( member(cut,Set) -> ! ; true ),
19        prove(Cla,Path,PathLim,[Lit|Lem],Set,Proof2).
```

Fig. 2. The prove function of leanCoP.

comparison with the proving process. The matrix and an initial clause are the arguments to the C prove function. In our implementation these are prepared by the higher-level code originating from HOL Light [4].

Term representation In order to achieve an efficient low-level algorithm, we start with a term representation with full sharing. As we do not use discrimination trees, flatterms were not considered. Each term consists of a tag and an array of pointers to term arguments. The tag stores a 32-bit signed integer and the length of the argument array. Negative tags represent functions and constants, while positive ones represent variables. Terms are transmitted to the low-level implementation bottom-up. The chosen term representation is similar to that of Prolog implementations [17,1] or E-prover [13][2].

As we want to preserve full sharing also in the presence of renaming, we introduce term offsets. Each first-order literal or clause will store its term arguments together with an integer *offset*, which represents a value that is implicitly added to all the variables in the literal or clause. All algorithms that operate on terms, literals, and clauses will need to compute variable offsets. We also implement full sharing for clauses: each clause is a reference to an array of literals, together with the position in this array. As literals are solved, only the position and offset are changed, no clause copies are required.

[2] In E-Prover it is the negative indices rather than positive ones that represent variables. We chose to use positive ones for variables, since they can directly be used as indices in the substitution array.

Global substitution One of the differentiating features of connection tableaux proof search algorithmsis the fact, that a single global substitution suffices. As unification produces new variable assignments, these are added to the global substitution. Similarly, when backtracking, a certain number of most recent assignments is removed. We represent the global substitution in a way, that allows all substitution operations (addition of an assignment, lookup, and retracting a most recent assignment) in constant time. To do so, we use an array and a stack (the stack is implemented as an array and an integer). The array stores pairs of terms and integer offsets and represents the actual substitution. The stack remembers the variables (integers) that have been assigned most recently. In order to add an assignment to the substitution, the array is updated and the assigned variable is added to the stack. To backtrack the assignment, the most most recent index is popped from the stack and this assignment is removed from the array.

Equality and Unification Checking the equality of terms with offsets under the substitution is straightforward: A helper stack stores pairs of terms (together with offsets) that need to be checked for equality. When a pair of terms is popped from the stack, the encountered variables are resolved in the substitution. If the two are applications of the same function symbol, the pointers to the arguments are pushed on the stack.

In a similar way we implement Robinson's unification algorithm [12] for terms with offsets. It is known to perform very well for practical first-order theorem proving [2]. The offsets allow our single implementation to cover both unification with and without renaming.

Prolog backtracking To implement the semantics of Prolog backtracking we use two stacks. We call these stacks *alternatives* and *promises*. The alternatives stack keeps all the possible backtracking points, while the promises stack keeps the information about the calls to `prove` that need to be done after our current one is successful. We improve on the idea of using two stacks which we presented before [4] by making the code tail-recursive and ensuring that no memory needs to be allocated.

Each alternative entry stores a tuple consisting of a pointer to a clause, the number of entries on the path, the number of lemmas, the number of substitution entries, the number of promises, and the actual number of the alternative matrix entry. To represent the path, lemmas, and substitution it suffices to store an integer that represents the size of each respective stack. Storing an alternative consists of storing the current pointers and numbers to the stack of alternatives and can be done in constant time. Whenever the current goal fails, we pop an alternative from the stack, change the state to match that saved in the alternative entry and call the prove function. In order to restore the state, apart from changing the integer variables, the path, lemmas, and substitution need to be restored. In case of the path and the lemmas, it is enough to update the stack size: no array updates are needed. In case of the substitution, the given number of most recent entries need to be removed from the array. Since each entry in

the substitution array needed to be added in constant time, the whole operation of switching to an alternative can be done in constant amortized time.

Each promise entry keeps a tuple consisting of a clause, path, lemmas, new lemma, and the number of alternatives. Once again the path, lemmas, and alternatives can be stored as single integers. As the goal succeeds, when switching to the next promised goal, we also need to realize the cut after an extension step. This is done by forgetting a number of most recent alternatives and can be done by updating the size of the alternatives stack to the stored one.

Core `prove` function The code of the core prove loop (equivalent of the Prolog `prove` function) is presented in Fig. 3. The backtracking mechanism needs to be able to switch back to three different parts of the function (checking for regularity, reduction, and extension steps). Thus, we implement the prove function as three functions that call each other recursively: `prove`, `extend`, and `reduce`. Sibling call optimization implemented by modern C compilers produces code that does not allocate stack frames. The three functions return a boolean, that indicates whether the proposition was proved, if so the proof can be inspected in the global arrays.

The `prove` function first checks if the clause is empty (line 2). If so, we can proceed with the promises. Next, if there is the same literal in the clause and on the path we continue with an alternative (lines 3–6). Finally, if there is a lemma that matches the current literal, we continue with the rest of the clause (lines 7–11) otherwise we continue to the `reduce` function (line 12).

The `reduce` function takes a starting position in the path as an argument. It tests all the literals on the path starting at the given position for unification with the negated literal. If successful, we continue by a recursive call to the `prove` function with the rest of the clause, additionally storing a backtracking point (lines 17–19). The backtracking point stores the information that it should call reduce with the index of the next literal on the path. If no path literal unifies, we proceed to the `extend` function (line 22).

The `extend` function iterates over all matrix entries matching the negated predicate symbol, starting entry given as the argument (lines 25–40). It first checks for iterative deepening termination condition if the clause is non-ground (line 27). Next it tries to unify the literal with the matrix clause. If successful, it stores a backtracking point in the alternatives and the rest of the clause as a promise (lines 29–30) and continues with the rest of the matrix clause. Renaming of the clause is performed by changing the offset value. If `extend` did not find a clause that would unify with the literal, we backtrack to an alternative.

4 Evaluation

To compare the efficiency of the low-level implementation with a Prolog one, we made sure that the implementations start with the same CNF. As leanCoP's Prolog parser can only parse FOF problems, and only those which contain at most one conjecture, we selected all the CNF problems in TPTP version 6.0.0 that contain precisely one conjecture and transformed them to FOF using `tptp4X`.

```
1  bool prove() {
2    if (cl_start == cl_len) return try_promise();
3    for (int i = cl_start; i < cl_len; ++i)
4      for (int j = 0; j < path_len; ++j)
5        if (lit_eq(path[j], cl[i], cl_off))
6          return try_alternative();
7    for (int i = 0; i < lem_len; ++i)
8      if (lit_eq(lem[i], cl[cl_start], cl_off)) {
9        cl_start++;
10       return prove();
11     }
12   return reduce(path_len - 1);
13 }
14 bool reduce(int n) {
15   for (; n >= 0; --n) {
16     if (neg_unify(path[n].t,path[n].o, cl[cl_start], cl_off)) {
17       cl_start++;
18       return prove();
19     }
20   }
21   return extend(0);
22 }
23 bool extend(int i) {
24   int pred = negate(cl[cl_start]->f);
25   for (; i < db_len[pred]; ++i) {
26     struct db_entry dbe = db[pred][i];
27     if (path_len >= path_lim && dbe.vars > 0) continue;
28     if (lit_unify(cl[cl_start], cl_off, dbe.lit1, sub_off)) {
29       store_alternative(true, i+1, old_sub);
30       store_promise();
31       path[path_len].t = cl[cl_start];
32       path[path_len++].o = cl_off;
33       cl = dbe.rest;
34       cl_start = 0;
35       cl_len = dbe.rest_len;
36       cl_off = sub_off;
37       sub_off += dbe.vars;
38       return prove();
39     }
40   }
41   return try_alternative();
42 }
```

Fig. 3. The core of the C implementation consists of three functions: `prove` checks regularity and lemmas, while `reduce` and `extend` implement the corresponding rules.

Additionally, to make sure the order of the equality axioms is the same, we used tptp4X to include the equality axioms, and changed the name of the equality predicate. We modified the source code of leanCoP 2.1 in two ways: only one

TPTP	Number of inferences			Maximum depth			number of
Category	low-level	Prolog	ratio	low-level	Prolog	ratio	problems
ALG	7.54	0.25	29.57	11.13	9.38	1.19	55
BOO	7.06	0.57	12.38	28.29	22.54	1.26	100
COL	8.80	0.76	11.64	26.13	20.16	1.30	119
GEO	14.90	0.38	39.38	6.83	5.65	1.21	118
GRP	10.85	0.78	13.94	19.59	16.67	1.18	593
LAT	3.80	0.51	7.39	27.58	23.57	1.17	259
LCL	6.01	0.43	13.96	155.60	88.97	1.75	454
NUM	17.73	0.14	127.60	13.95	8.73	1.60	94
RNG	12.55	0.89	14.13	19.07	16.28	1.17	76
SET	13.91	0.20	68.50	7.22	6.04	1.19	260
SWW	16.67	0.74	22.60	11.08	9.21	1.20	71
all	9.84	0.54	18.37	39.39	25.93	1.52	2936

Table 1. Number of inferences (in millions) and maximum depth (non-ground path length limit) reached by the two implementations in 60 seconds averaged over each TPTP category with at least 50 problems and averaged over all TPTP problems. Only the problems for which cut does not yield a proof are considered. The full version of the table with all individual problems and all categories is available at: `http://cl-informatik.uibk.ac.at/users/cek/tableaux15/`

strategy is selected and literals in clauses are not reordered. We chose to focus on the `[cut, conj, nodef]` strategy. The evaluations have been done on a server with 48 AMD Opteron 6174 2.2 GHz CPUs, 320 GB RAM and 0.5 MB L2 cache per CPU. Each ATP problem is assigned a single core. In an initial evaluation we tested SWI-Prolog version 6.6.6 against ECLiPSe 5.10. With the former being able to solve 2 more of the problems, we focus on it in all further evaluations in the paper. Similarly we compared GCC 4.9 against Clang 3.5, again with a small advantage of the former.

We patched both implementations to increase an inference counter at every extension step and print the number of inferences and the maximum path length at every iterative deepening step. We show the average numbers of inferences for all the problems that were not solved in 60 seconds by either of the implementations in Table 1 averaged by TPTP category and globally. We focus on the non solved problems, as for the solved ones the numbers of inferences and the depth of the proofs are same. For a small number of problems in geometry (`GEO001-4`, `GEO002-4`) the numbers of inferences are the same for the two implementations, however in the majority of problems, the low-level implementation is able to perform significantly more inferences, with the biggest difference for `GRP015-1`: the low-level code can perform 31 million inferences, while the Prolog code can do only 3,341 inferences. This directly corresponds to reaching a higher maximum path length in the iterative deepening: the low-level implementation can reach a path length that is on average 52% longer than the Prolog one.

A different way in which the performance of the two implementation can be compared, is to directly look at the numbers of solved problems. This is done

Implementation	Theorems	Unique
low-level	693	37
Prolog	656	0

Table 2. Numbers of problems solved by the two implementations in 60 seconds.

in Table 2, the low-level version solved 37 new TPTP problems, which is 5.6% more than the Prolog one.

5 Conclusion

We implemented the core of the leanCoP theorem prover, together with its cut semantics, in C. We optimized the representations of terms and substitution, as well as the algorithms of equality checking and unification in the implementation. We evaluated the efficiency of the generated C code against the Prolog code on on a large subset of TPTP CNF problems. We were surprised by the difference: The low-level implementation can on average perform 18 times more inferences per second than the Prolog one. This corresponds to 5.6% more TPTP problems solved by a single strategy.

The C implementation is reasonably concise, totalling 350 lines of code. This includes 42 LoC in the core `prove` function, 61 LoC implementing Prolog backtracking, and 172 LoC for the shared terms, clauses, and unification. The C implementation together with the high-level parsing code and the complete statistics are available at:

`http://cl-informatik.uibk.ac.at/~cek/tableaux15`.

The use of imperative data structures allows random access. This means further optimizations and experiments are possible, that would be hard to achieve in Prolog, such as:

- backtracking points can be introduced only when needed (for example if a unification returns a non-empty substitution, or if it is more general than a previous one);
- the path can be traversed in the opposite direction, changing the introduced alternatives;
- reordering of literals in clauses, as done by randoCoP [11], can be done completely in place.
- cut can remove a different number of backtracking points, than that specified by the semantics of the Prolog cut, which could give rise to half-cut or double-cut strategies;

Other future work ideas involve the integration of some of the advanced strategies for tableaux proving, such as those implemented in SETHEO [8], or combining our implementation with fast low-level machine learning algorithms [3] for internal proof guidance [16].

References

1. K. R. Apt and M. Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2007.
2. K. Hoder and A. Voronkov. Comparing unification algorithms in first-order theorem proving. In B. Mertsching, M. Hund, and M. Z. Aziz, editors, *KI 2009: Advances in Artificial Intelligence*, volume 5803 of *LNCS*, pages 435–443. Springer, 2009.
3. C. Kaliszyk, S. Schulz, J. Urban, and J. Vyskočil. System description: E.T. 0.1. In *Conference on Automated Deduction*, LNCS. Springer Verlag, 2015. to appear.
4. C. Kaliszyk, J. Urban, and J. Vyskočil. Certified connection tableaux proofs for HOL Light and TPTP. In X. Leroy and A. Tiu, editors, *Proc. of the 4th Conference on Certified Programs and Proofs (CPP'15)*, pages 59–66. ACM, 2015.
5. K. Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.
6. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
7. R. Letz and G. Stenz. Model elimination and connection tableau procedures. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 2015–2114. Elsevier and MIT Press, 2001.
8. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – the CADE-13 systems. *J. Autom. Reasoning*, 18(2):237–246, 1997.
9. J. Otten. Restricting backtracking in connection calculi. *AI Commun.*, 23(2-3):159–182, 2010.
10. J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003.
11. T. Raths and J. Otten. randocop: Randomizing the proof search order in the connection calculus. In B. Konev, R. A. Schmidt, and S. Schulz, editors, *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning, Sydney, Australia, August 10-11, 2008*, volume 373 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
12. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
13. S. Schulz. System description: E 1.8. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.
14. G. Sutcliffe. The 4th IJCAR automated theorem proving system competition - CASC-J4. *AI Commun.*, 22(1):59–72, 2009.
15. G. Sutcliffe. The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Commun.*, 26(2):211–223, 2013.
16. J. Urban, J. Vyskočil, and P. Štěpánek. MaLeCoP: Machine learning connection prover. In K. Brünnler and G. Metcalfe, editors, *TABLEAUX*, volume 6793 of *LNCS*, pages 263–277. Springer, 2011.
17. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *TPLP*, 12(1-2):67–96, 2012.