lolliCoP – A Linear Logic Implementation of a Lean Connection-Method Theorem Prover for First-Order Classical Logic

Joshua S. Hodas and Naoyuki Tamura*

Department of Computer Science
Harvey Mudd College
Claremont, CA 91711, USA
hodas@cs.hmc.edu,tamura@kobe-u.ac.jp

Abstract. When Prolog programs that manipulate lists to manage a collection of resources are rewritten to take advantage of the linear logic resource management provided by the logic programming language Lolli, they can obtain dramatic speedup. Thus far this has been demonstrated only for "toy" applications, such as n-queens. In this paper we present such a reimplementation of the lean connection-calculus prover leanCoP and obtain a theorem prover for first-order classical logic which rivals or outperforms state-of-the-art provers on a significant body of problems.

1 Introduction

The development of logic programming languages based on intuitionistic [11] and linear logic [6] has been predicated on two principal assumptions. The first, and the one most argued in public, has been that, given the increased expressivity, programs written in these languages are more perspicuous, more natural, and easier to reason about formally. The second assumption, which the designers have largely kept to themselves, is that by moving the handling of various program features into the logic, and hence from the term level to the formula level, we would expose them to the compiler, and, thus, to optimization. In the end, we believed, this would yield programs that executed more efficiently than the equivalent program written in more traditional logic programming languages. Until now, this view has been downplayed as most of these new languages have thus far been implemented only in relatively inefficient, interpreted systems.

With the recent development of compilers for languages such as λ-Prolog [13] and Lolli [7], however, we are beginning to see this belief justified. In the case of Lolli, we are focused on logic programs which have used a term-level list as a sort of bag from which items are selected according to some rules. In earlier work we showed that when such code is rewritten in Lolli, allowing the elements in the list to instead be stored in the proof context –with the underlying rules

* This paper reports work done while the second author was on a sabbatical-leave from Kobe University.

of linear logic managing their consumption– substantial speedups can occur. To date, however, that speedup has been demonstrated only on the execution of simple, “toy” applications, such as an n-queens problem solver [7].

Now we have turned our attention to a more sophisticated application: theorem proving. We have reimplemented the leanCoP connection-calculus theorem prover of Otten and Bibel [14] in Lolli. This “lean” theorem prover has been shown to have remarkably good performance relative to state-of-the-art systems, particularly considering that it is implemented in just a half-page of Prolog code. The reimplemented prover, which we call lolliCoP, is of comparable size, and, when compiled under LLP (the reference Lolli compiler [7]), provides a speedup of 40% over leanCoP. On many of the hardest problems that both can solve, it is roughly the same speed as the OTTER theorem prover [8]. (Both leanCoP and lolliCoP solve a number of problems that OTTER cannot. Conversely, OTTER solves many problems that they cannot. On simpler problems that both solve, Otter is generally much faster than leanCoPand lolliCoP.)

While this is a substantial improvement, it is not the full story. LLP is a relatively naive, first-generation compiler and run-time system. Whereas, it is being compared to a program compiled in a far more mature and optimized Prolog compiler (SICStus Prolog 3.7.1). When we adjust for this difference, we find that lolliCoP is more than twice as fast as leanCoP, and solves (within a limited time allowance) more problems from the test library. Also, when the program is rewritten in Lolli, two simple improvements become obvious. When these changes are made to the program, performance improves by a further factor of three, and the number of problems solved expands even further.

1.1 Organization

The remainder of this paper is organized as follows: Section 2 gives a brief introduction to the connection calculus for first-order classical logic; Section 3 describes the leanCoP theorem prover; Section 4 gives a brief introduction to linear logic, Lolli, and the LLP compiler; Section 5 introduces lolliCoP; Section 6 presents the results and analysis of various performance tests and comparisons; and, Section 7 presents the two optimizations mentioned above.

2 Connection-Calculus Theorem Proving

The connection calculus [2] is a matrix proof procedure for clausal first-order classical logic. (Variations have been proposed for other logics, but this is its primary application.) The calculus, which uses a positive representation, proving matrices of clauses in disjunctive normal form, has been utilized in a number of theorem proving systems, including KoMeT [3], SETHEO and E-SETHEO [9, 12]. It features two principal rules, *extension* and *reduction*. The *extension* step, which corresponds roughly to backchaining, consists of matching the complement of a literal in the active goal clause with the head of some clause in the matrix. The body of that clause is then proved, as is the remainder of the original clause.

$$\frac{\Gamma \mid \emptyset \vdash A_1, \ldots, A_n}{\Gamma} \ (start) \qquad \frac{}{\Gamma \mid \Pi \vdash} \ (extension_0)$$

(provided $C \in \Gamma$, $C[\boldsymbol{t}/\boldsymbol{x}] = \{A_1, \ldots, A_n\}$ for some $\boldsymbol{t}$, $n \geq 0$)

$$\frac{\Gamma \mid L_i, \Pi \vdash L_{11}, \ldots, L_{1m} \quad C, \Gamma \mid \Pi \vdash L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n}{C, \Gamma \mid \Pi \vdash L_1, \ldots, L_n} \ (extension_1)$$

(provided C is ground, $C = \{\overline{L_i}, L_{11}, \ldots, L_{1m}\}$, $1 \leq i \leq n$, and $m \geq 0$)

$$\frac{C, \Gamma \mid L_i, \Pi \vdash L_{11}, \ldots, L_{1m} \quad C, \Gamma \mid \Pi \vdash L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n}{C, \Gamma \mid \Pi \vdash L_1, \ldots, L_n} \ (extension_2)$$

(provided C is not ground, $C[\boldsymbol{t}/\boldsymbol{x}] = \{\overline{L_i}, L_{11}, \ldots, L_{1m}\}$ for some $\boldsymbol{t}$, $1 \leq i \leq n$, and $m \geq 0$)

$$\frac{\Gamma \mid \overline{L_i}, \Pi \vdash L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n}{\Gamma \mid \overline{L_i}, \Pi \vdash L_1, \ldots, L_n} \ (reduction, 1 \leq i \leq n)$$

Fig. 1. A deduction system for the derivation relation of the Connection Calculus

For the duration of the proof of the body of the matching clause, however, the literal that matched is added to a secondary data structure called the *path.* If at a later point the complement of a literal being matched occurs in the path, that literal need not be proved. This short-circuiting of the proof constitutes the *reduction* step. Search terminates when the goal clause is empty. Finally, note that in the *extension* step, if the clause matched is ground, it is removed from the matrix during the subproof.

Figure 1 shows a deduction system for the derivation relation. Two versions of the *extension* rule are given, depending on whether the matched clause is ground or not. A third version handles the termination case. In the core rules of this system, the left-hand side of the derivation has two parts: the matrix, Γ, is a multiset of clauses; the path, Π, is a multiset of literals. The goal clause on the right-hand side is a sequence of literals. Note that the calculus is more general than necessary. We can, without loss of completeness, restrict the selection of a literal from the goal clause to the leftmost literal (i.e., restrict $i = 1$).

A derivation is a deduction tree rooted at an application of the *start* rule, for some positive clause C, with instances of *extension*$_0$ and premiseless instances of *reduction* at the leaves. In an implementation, the choice of terms $\boldsymbol{t}$ in the *start* and *extension*$_2$ rules would be delayed via unification in the usual manner. Otten and Bibel provide an alternate, isomorphic, formulation of the calculus by way of an operational semantics in which substitutions are made explicit [14].

3 The leanCoP Theorem Prover

The leanCoP theorem prover of Otten and Bibel [14] is a Prolog program, shown in Figure 2, providing a direct encoding of the calculus shown in Figure 1. In this implementation clauses, paths, and matrices are represented as Prolog lists. Atomic formulas are represented with Prolog terms. A negated atom is represented by applying the unary `-` operator to the corresponding term. Prolog

```
prove(Mat) :- prove(Mat,1).

prove(Mat,PathLim) :-
    append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
    append(MatA,MatB,Mat1), prove([!],[[-!|Cla]|Mat1],[],PathLim).
prove(Mat,PathLim) :-
    \+ground(Mat), PathLim1 is PathLim+1, prove(Mat,PathLim1).

prove([],_,_,_).
prove([Lit|Cla],Mat,Path,PathLim) :-
    (-NegLit=Lit; -Lit=NegLit) ->
    ( member_oc(NegLit,Path) ;
      append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
      append_oc(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
      ( Cla1==Cla2 -> append(MatB,MatA,Mat1)
                    ; length(Path,K), K<PathLim,
                      append(MatB,[Cla1|MatA],Mat1)
      ), prove(Cla3,Mat1,[Lit|Path],PathLim)
    ), prove(Cla,Mat,Path,PathLim).
```

Fig. 2. The leanCoP theorem prover of Otten and Bibel

variables are used to represent object variables. This last fact causes some complications, discussed below.

The first evident difference between the calculus and its implementation is that an extra value, an integer path-depth limit, is added to each of the Prolog predicates. It is used to implement iterative deepening based on the maximum allowed path length, which is necessary to insure completeness in the first-order case, due to Prolog's depth-first search strategy. When `prove/1` is called, it sets the initial path limit to 1 and calls `prove/2`, which in turn selects (without loss of generality) a purely positive start clause.

The selection of the clause, `Cla`, is done using a trick of Prolog: since the predicate `append(A,B,C)` holds if the list `C` results from appending list `B` to list `A`, `append(A,[D|B],C)` (in which `[D|B]` is a list that has `D` as it's first item, followed by the list `B`) will hold if `D` is an element of `C` and if, further, `A` is the list of items preceding it and `B` is the list of items following it. Thus Prolog can, in one predicate, select an element from an arbitrary position in a list and identify all the remaining elements in the list, which result from appending `A` and `B`.

This technique is used to select literals from clauses and clauses from matrices throughout leanCoP. While it is an interesting trick, it relies on significant manipulation and construction of list structures on the heap. It is precisely certain uses of this trick which will be replaced by linear logic resource management at the formula level in lolliCoP.

To insure that the selected clause is purely positive, the code checks that the clause contains no negated terms (terms of the form `-_`, where the underscore is a wildcard). This is done using Prolog's negation-as-failure operator: `\+`. Once this is confirmed, the proof is started using a dummy (unit) goal clause, `!`, which will cause the selected clause to become the goal clause in the next step. This is done to avoid duplicating some bookkeeping code already present in the general case in `prove/4`, which implements the core of the prover. Note that the similarity of appearance to the Prolog cut operator is coincidental.

Should the call to `prove/4` at the end of the first clause of `prove/2` fail, then, provided this is not a purely propositional problem (That is, if it is not true that the entire matrix is ground.) the second clause of `prove/2` will cause the entire process to repeat, but with a path-depth limit one larger.

The first clause of `prove/4` implements the termination case, $extension_0$, and is straightforward. The second implements the remaining rules. This clause begins by selecting, without loss of completeness, the first literal, `Lit`, from the goal clause. If the complement of this literal as computed by the first line of the body of the clause matches a literal in the `Path`, then the system attempts to apply an instance of the *reduction* rule, jumping to the last line of the clause, where it recursively proves the remainder of the goal using the same matrix and path, under the substitution resulting from the matching process. (That is, free variables in literals in the goal and the path may have become instantiated.)

If a match to the complement of the literal is not found on the path, that is, if all attempts to apply instances of *reduction* have failed, then this is treated as either $extension_1$ or $extension_2$, depending on whether or not the clause selected next is ground. A clause is selected by the technique described above. Then a literal matching the complement of the goal literal is selected from the clause. (If this fails then the program backtracks and selects another clause.) The test `Cla1==Cla2` is used, as explained below, to determine if the selected clause is ground, and the matrix for the subproof is constructed accordingly, either with or without the chosen clause. If the path limit has not been reached, the prover recursively proves the body of the selected clause under the new path assumption and substitution, and, if it succeeds, goes on to prove the remainder of the current goal clause. As the depth-first prover is complete for propositional logic, the path limit check is not done if the selected clause is ground.

Note, `P -> Q ; R` is an extra-logical control structure corresponding to an `if-then-else` statement, The difference between this and `((P,Q) ; (\+P,R))` is that the latter allows for backtracking and retrying the test under another substitution, whereas the former allows the test to be computed only once and an absolute choice is made at that point. It can also be written without `R`, as is done in some cases here. Such use is, in essence, a hidden use of the Prolog cut operator, which is used for pruning search.

As mentioned above, the use of Prolog terms to represent atomic formulas introduces complications. This is because the free variables of a term, intended to represent the implicitly quantified variables of the atoms, can become bound if the term is compared (unified) with another term. In order to avoid the variables in clauses in the matrix from being so bound, when a clause is selected from the matrix, a copy with a fresh set of variables is produced using `copy_term`, and that copy is the clause that is used. Thus, the comparison `Cla1==Cla2`, which checks for syntactic identity, succeeds only if there were no variables in the

$$\frac{}{\Gamma; B \longrightarrow B}\ \textit{identity} \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ \top_R \qquad \frac{}{\Gamma; \emptyset \longrightarrow 1}\ 1_R$$

$$\frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ \multimap_R \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ \Rightarrow_R$$

$$\frac{\Gamma; \Delta \longrightarrow B \quad \Gamma; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \mathbin{\&} C}\ \&_R \qquad \frac{\Gamma; \Delta_1 \longrightarrow B_1 \quad \Gamma; \Delta_2 \longrightarrow B_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow B_1 \otimes B_2}\ \otimes_R$$

$$\frac{\Gamma; \Delta \longrightarrow B[x \mapsto t]}{\Gamma; \Delta \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Gamma; \Delta \longrightarrow B_i}{\Gamma; \Delta \longrightarrow B_1 \oplus B_2}\ \oplus_{R_i}$$

$$\frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ \textit{absorb} \qquad \frac{\Gamma; \Delta, B[x \mapsto t] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ \forall_L$$

$$\frac{\Gamma; \emptyset \longrightarrow B \quad \Gamma; \Delta, C \longrightarrow E}{\Gamma; \Delta, B \Rightarrow C \longrightarrow E}\ \Rightarrow_L \qquad \frac{\Gamma; \Delta_1 \longrightarrow B \quad \Gamma; \Delta_2, C \longrightarrow E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \longrightarrow E}\ \multimap_L$$

Fig. 3. A proof system for a fragment of linear logic

original term `Cla1` (since they would have been modified by `copy_term`), and, hence, if that term was ground.

Because Prolog unification is unsound, as it lacks the "occurs check" for barring the construction of cyclic unifiers, if the prover is to be sound we must force sound unification when comparing literals. In Eclipse Prolog, used in the original leanCoP paper, this is done with a global switch, affecting all unification in the system. In SICStus Prolog, used for the tests in this paper, it is done with the predicate `unify_with_occurs_check`. This predicate is used within the `member_oc` and `append_oc` predicates, whose definitions have been elided in the code above.

Many of these complications could have been avoided by using λ-Prolog, which supports the use of λ-terms as data for representing name-binding structures, and whose unification algorithm is sound [11].

4 A Brief Introduction to Linear Logic Programming

Linear logic was first proposed by Girard in 1987 [4]. Figure 3 gives a Gentzen sequent calculus for part of the fragment of intuitionistic linear logic which forms the foundation of the logic programming language Lolli, named for the linear logic implication operator, $\multimap$, known as lollipop. The calculus is not the standard one, but for this fragment is equivalent to it, and is easier to explain in the context of logic programming. In these sequents, the left-hand side has two parts: the context Γ holds assumptions that can be freely reused and discarded, as in traditional logics, while the assumptions in Δ, in contrast, must be used exactly once in a given branch of a tree. The two implication operators, $\Rightarrow$, and $\multimap$, are used to add assumptions to the unrestricted and linear contexts, respectively. In Lolli they are written `=>` and `-o`.

In the absence of contraction and weakening (that is, the ability to freely reuse or discard assumptions, respectively), all of the other logical operators split into two variants as well. For example, the conjunction operator splits into *tensor*, $\otimes$, and *with*, &. In proving a conjunction formed with $\otimes$, the current set of restricted assumptions, Δ, is split between the two conjuncts: those not used in proving the first conjunct must be used while proving the second. To prove a & conjunction, the set of assumptions is copied to both sides: each conjunct's proof must use all of the assumptions. In Lolli, the $\otimes$ conjunction is represented by the familiar "`,`". This is a natural mapping, as we expect the effect of a succession of goals to be cumulative: each has available to it the resources not yet used by its predecessors. The & conjunction, which is less used, is written "`&`".

Thus, a query showing that two dollars are needed to buy pizza and soda when each costs a dollar can be written in Lolli as:

```
?- (dollar -o pizza) => (dollar -o soda) =>
          (dollar -o dollar -o (pizza,soda))
```

which would succeed. In contrast, a single, ordinary dollar would be insufficient, as in the failing query:

```
?- (dollar -o pizza) => (dollar -o soda) =>  (dollar -o (pizza,soda))
```

If we wished to allow ourselves a single, infinitely reusable dollar, we would write:

```
?- (dollar -o pizza) => (dollar -o soda) =>  (dollar => (pizza,soda))
```

which would also succeed. Finally, the puzzling query:

```
?- (dollar -o pizza) => (dollar -o soda) =>  (dollar -o (pizza & soda))
```

would also succeed. It says that with a dollar it is possible to buy soda and possible to buy pizza, but not both at the same time.

It is important to note that while the implication operators add clauses to a program while it is running, they are not the same as the Prolog `assert` mechanism. First, the addition is scoped over the subgoal on the right of the implication, whereas a clause `assert`ed in Prolog remains until it is `retract`ed. So, for example, the following query will fail:

```
?- (dollar => dollar), dollar.
```

Assumed clauses also go out of scope if search backtracks out of the subordinate goal. Second, whereas `assert` automatically universalizes any free variables in an added clause, in Lolli clauses added with implication can contain free logic variables, which may get bound when the clause is used to prove some goal. Therefore, whereas the Prolog query:

```
?- assert(p(X)), p(a), p(b).
```

will succeed, because `X` is universalized, the seemingly similar Lolli query:

```
?- p(X) => (p(a), p(b)).
```

will fail, because the attempt to prove `p(a)` causes the variable `X` to become instantiated to `a`. If we desire the other behavior, we must quantify explicitly:

```
?- (forall X\p(X)) => (p(a), p(b)).
```

What's more, any action that causes the variable `X` to become instantiated will affect instances of that variable in added assumptions. For example, the query:

```
?- p(X) => r(a) => (r(X), p(b)).
```

will fail, since proving `r(X)` causes the variable `X` to be instantiated to `a`, both in that position, and in the assumption `p(X)`. Our implementation of lolliCoP will rely crucially on all these behaviors.

Though there are two forms of disjunction in linear logic, only one, "$\oplus$" is used in Lolli. It corresponds to the traditional one and is therefore written with a semicolon in Lolli as in Prolog.

There are also two forms of truth, $\top$, and 1. The latter, which Lolli calls "`true`", can only be proved if all the linear assumptions have already been used. In contrast, $\top$ is provable even if some resources are, as yet, unused. Thus if a $\top$ occurs as one of the conjuncts in a $\otimes$ conjunction, then the conjunction may succeed even if the other conjuncts do not use all the linear resources. The $\top$ is seen to consume the leftovers. Therefore, Lolli calls this operator "`erase`".

It is beyond the scope of this paper to demonstrate the applications of all these operators. Many good examples can be found in the literature, particularly in the papers on Lygon and Lolli [5, 6]. The proof theory of this fragment has also been developed extensively [6]. Of crucial importance is that there is a straightforward goal-directed proof procedure (conceptually similar to the one used for Prolog) that is sound and complete for this fragment of linear logic.

5 The lolliCoP Theorem Prover

Figure 4 gives the code for lolliCoP, a reimplementation of leanCoP in Lolli/LLP.[1] The basic premise of its design is that, rather than being passed around as a list, the matrix will be loaded as assumptions into the proof context and accessed directly. In addition, ground clauses will be added as linear resources, since the calculus dictates that in any given branch of the proof, a ground clause should be removed from the matrix once it is used. Non-ground clauses are added to the intuitionistic (unbounded) context. In either case (ground or non-ground) these assumptions are stored as clauses for the special predicate `cl/1`. Literals in the path are also stored as assumptions added to the program. They are unbounded assumptions added as clauses of the special predicate `path`. While Lolli supports the λ-terms of λ-Prolog, LLP does not. Therefore, clauses are still represented as lists of literals, which are represented as terms as before.

The proof procedure begins with a call to `prove/1` with a matrix to be proved. This predicate first reverses the order of the clauses, so that when they are added recursively the resultant context will be searched in their original order. It then calls `pr/1` to load the matrix into the unrestricted and linear proof contexts, as appropriate. First, however, it checks whether the entire matrix is ground

[1] Because the LLP parser is written in Prolog, LLP uses `-<>` for $\multimap$, rather than `-o`.

```
prove(Mat) :- reverse(Mat,Mat1),
              (ground(Mat) -> propositional => pr(Mat1)
                            ; pr(Mat1)
              ).

pr([])        :- p(1).
pr([Cla|Mat]) :- (ground(Cla) -> (cl(Cla) -<> pr(Mat))
                               ; (cl(Cla)  => pr(Mat))
                 ).

p(PathLim) :- cl(Cla), \+member(-_,Cla),
              copy_term(Cla,Cla1), prove(Cla1,PathLim).

p(PathLim) :- \+propositional,
              PathLim1 is PathLim+1, p(PathLim1).

prove([],_) :- erase.
prove([Lit|Cla],PathLim) :-
  (-NegLit=Lit; -Lit=NegLit) ->
    ( path(NegLit), erase ;
      cl(Cla1), copy_term(Cla1,Cla2), append(ClaA,[NegLit|ClaB],Cla2),
      append(ClaA,ClaB,Cla3), (Cla1==Cla2 -> true ; PathLim>0),
      PathLim1 is PathLim-1, path(Lit) => prove(Cla3,PathLim1)
    ) & prove(Cla,PathLim).
```

Fig. 4. The lolliCoP theorem prover

or not. If it is, a flag predicate is assumed (using `=>`) to indicate that this is a propositional problem, and that iterative deepening is not necessary.

The predicate `pr/1` takes the first clause out of the given matrix, adds it to the current context as either a linear or unlimited assumption, as appropriate, and then calls itself recursively as the goal nested under the implication. Thus, each call to this predicate will be executed in a context which contains the assumptions added by all the previous calls. When the end of the given matrix is reached, the first clause of `pr/1` calls `p/1` with an initial path-length limit of 1, so that a start clause can be selected, and the proof search begun.

The clauses for `p/1` take the place of the clauses for `prove/2` in leanCoP. They are responsible for managing the iterative deepening, and for selecting the start clause for the search. A clause is selected just by attempting to prove the predicate `cl/1` which will succeed by matching one of the clauses from the matrix which has been added to the program. This is significantly simpler than the process in leanCoP. Once the program finds a purely positive start clause, it is copied and its proof is attempted at the current path-length limit. Should that process fail for all possible choices of start clause, the second clause of `p/1` is invoked. It checks to see that this is not a purely propositional problem, and if it is not, makes a recursive call with the path-length limit one higher.

The predicate `prove/2` takes the role of `prove/4` in leanCoP; because the matrix and path are stored in the proof context, they no longer need to be passed around as arguments. The first clause, corresponding to $extension_0$, here has a body consisting of the `erase` ($\top$) operator. Its purpose is to discard any linear assumptions (i.e. ground clauses in the matrix) that were not used in this branch of the proof. This is necessary since we are building a prover for classical logic, in which assumptions can be discarded.

The second clause of this predicate is, as before, the core of the prover, covering the remaining three rules. It begins by selecting a literal from the goal clause and forming its complement. If a literal matching the complement occurs as an argument to one of the assumed `path/1` clauses, then this is an instance of the *reduction* rule and this branch is terminated. As with the $extension_0$ rule, `erase` is used to discard unused assumptions.

Otherwise, the predicate `cl/1` extracts a clause from the matrix, which is then copied and checked to see if it contains a match for the complement of the goal literal. If the clause is ground or if the path-length limit has not been reached, the current literal is added to the path and `prove/2` is called recursively as a subordinate goal (within the scope of the assumption added to the path) to prove the body of the selected clause.

If this was an instance of the *reduction* rule, or if it was an instance of $extension_1$ or $extension_2$ and the proof of the body of the matching clause succeeded, the call to `prove/2` finishes with a recursive call to prove the rest of the current goal clause. Because this must be done using the same matrix and path that were used in the other branch of the proof, the two branches are joined with a & conjunction. Thus the context is copied independently to the two branches.

It is important to notice that, other than checking whether the path-length limit has been reached, there is no difference between the cases when the selected clause is ground or not. If it was ground, it was added to the context using linear implication, and, since it has been used (to prove the `cl/1` predicate), it has automatically been removed from the program, and, hence, the matrix. Also, lolliCoP uses a different method for checking path length against the limit: the limit is simply decremented each time a literal is added to the path. This is done because there is no way to access the whole path to check its length, but has the advantage of being significantly more efficient as well.

It is also important to note that, as mentioned before, we rely on the fact that free variables in assumptions retain their identity as logic variables and may become instantiated subsequently. In particular, the literals added to the path may contain instances of free variables from the goal clause from which they derive. Anything which causes these variables to become instantiated will similarly affect those occurrences in these assumptions. Thus, this technique could not be implemented using Prolog's `assert` mechanism. In any case, `assert`ed clauses are generally not as fast as compiled ones.

6 Performance Analysis

We have tested lolliCoP on the 2200 clausal form problems in the TPTP library version 2.3.0 [15, 8]. These consist of 2193 problems known to be unsatisfiable (or valid using positive representation) and 7 propositional problems known to be satisfiable (or invalid). Each problem is rated from 0.00 to 1.00 relative to its difficulty. A rating of "?" means the difficulty is unknown. No reordering of clauses or literals has been done.

The tests were performed on a Linux system with a 550MHz Pentium III processor and 128M bytes of memory. The programs were compiled with version 0.50 of `LLP` which generated abstract machine code executed by an emulator written in C. The time limit for all proof attempts was 300 seconds.

Table 1. Overall performance of OTTER, leanCoP, and lolliCoP

	Total	OTTER	leanCoP	lolliCoP	lolliCoP_2
Solved	2200	1602 (73%)	810 (37%)	822 (37%)	880 (40%)
0 to < 1 second		1209	541	554	614
1 to < 10 seconds		142	135	124	117
10 to <100 seconds		209	93	91	94
100 to <200 seconds		31	18	25	34
200 to <300 seconds		11	23	28	21
Problems rated 0.00	1308	1230 (94%)	713 (55%)	716 (55%)	737 (56%)
Problems rated >0.00	733	249 (34%)	76 (10%)	83 (11%)	118 (16%)
Problems rated ?	159	123 (77%)	21 (13%)	23 (14%)	25 (16%)

The overall performance of OTTER 3.1 (with MACE 1.4) [8, 10], leanCoP [14], and lolliCoP, in terms of the number of problems solved, are shown in Table 1. The table also includes data for an improved version of lolliCoP, called lolliCoP_2, discussed in the next section. The results for leanCoP were obtained in the same environment as those for lolliCoP, using SICStus Prolog 3.7.1, and are better than those reported by the authors [14]. The results for OTTER 3.1 (with MACE 1.4), which is not publicly available, are taken from a report by its developers [8]. These results were produced on a 400MHz Pentium II, which is somewhat slower than the machine we used.

It is interesting to note that lolliCoP solved 57 problems, and lolliCoP_2 77, which OTTER can not solve. Most of these (47 for lolliCoP and 63 for lolliCoP_2) are rated higher than 0.00. It should also be noted that leanCoP solved ten problems that neither lolliCoP nor lolliCoP_2 solved. Since nine of these were rated 0.0, and given the structural similarities of the systems, we believe this to be due to serendipitous advantages with respect to clause ordering, since leanCoP orders clauses slightly differently. Fig. 5 depicts the overlap of problems solved by each system.

6.1 Performance comparison

In order to produce a more detailed comparison, we tested all the systems on the 118 problems rated greater than 0.0 which lolliCoP_2 can solve. Because OTTER

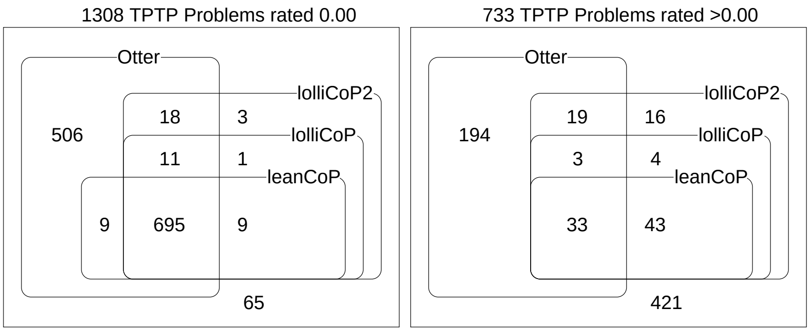

Fig. 5. Performance of OTTER, leanCoP and lolliCoP classified by problem rating

3.1 is not yet available, we used OTTER 3.0.6 instead. All tests were made on the same 550MHz Pentium III. Table 2 gives the results of this comparison. (OTTER results labeled "error" refer to an empty set-of-support.)

As mentioned in the introduction, although the table shows lolliCoP as almost consistently outpacing leanCoP these results do not tell the entire story. Because LLP is a first-generation implementation, the code generator is not nearly as sophisticated as SICStus', nor is its runtime system. To adjust for this factor we also executed a version of leanCoP using the LLP compiler and runtime system (since Lolli is a superset of Prolog). In this test, looking only at the problems that it succeeded in solving, leanCoP took 2.3 times as long as lolliCoP, providing a more accurate measure of the benefits accrued from the logical treatment.

Table 3a compares the performance of all four systems on the 33 problems that they can all solve. Total CPU time is shown, along with a speedup ratio relative to leanCoP (under SICStus). On just these problems, lolliCoP has almost the same performance as OTTER. However, comparing the result of 36 problems solved by both OTTER and lolliCoP, OTTER is 71% faster as shown in Table 3b. Finally, Table 3c shows a similar analysis for the 76 problems that lolliCoP and leanCoP can both solve.

7 Improvements to the lolliCoP Prover

In the design of leanCoP, Otten and Bibel seem to have been focused primarily on keeping the code as short as possible. In the process of reimplementing the system in Lolli, a simple but significant performance improvement became apparent, which we discuss here.

The most obvious inefficiency in the system as described thus far is that `copy_term` is called in order to create a new set of logic variables in a selected clause, even when the clause is ground, since that test is not made till later on. Given the size of some of the clauses in the problems in the TPTP library, this can be quite inefficient. While the obvious solution would be to move the use of `copy_term` into the body of the if-then-else along with the path-limit check, Lolli affords a more creative solution.

Table 2. Problems solved by lolliCoP$_2$ and rated higher than 0.00

Problem	Rating	Otter	leanCoP	lolliCoP	lolliCoP$_2$
BOO012-1	(0.17)	3.44	8.13	7.33	1.28
BOO012-3	(0.33)	17.39	237.81	63.00	9.43
CAT002-4	(0.17)	2.67	>300	>300	231.07
CAT003-2	(0.50)	>300	15.83	12.05	4.26
CAT003-3	(0.11)	>300	2.43	1.70	0.34
CAT012-4	(0.17)	0.26	19.95	14.88	4.57
COL002-3	(0.33)	>300	0.01	0.03	0.01
COL075-1	(0.50)	>300	>300	275.77	60.29
FLD002-3	(0.67)	1.20	201.23	162.01	43.73
FLD003-1	(0.67)	>300	>300	264.10	70.76
FLD004-1	(0.67)	>300	>300	>300	201.74
FLD009-3	(0.33)	>300	>300	272.94	72.59
FLD013-1	(0.67)	>300	0.46	0.48	0.15
FLD013-2	(0.67)	>300	>300	>300	106.83
FLD013-3	(0.33)	>300	>300	>300	153.85
FLD013-4	(0.33)	3.19	>300	>300	270.74
FLD016-3	(0.33)	11.90	>300	>300	155.71
FLD018-1	(0.33)	>300	>300	>300	101.86
FLD019-1	(0.33)	>300	>300	>300	196.80
FLD022-3	(0.33)	12.45	>300	>300	155.83
FLD023-1	(0.33)	>300	0.62	0.48	0.13
FLD025-1	(0.67)	>300	0.45	0.49	0.15
FLD025-3	(0.33)	>300	>300	>300	130.73
FLD028-3	(0.33)	13.45	>300	>300	187.66
FLD030-1	(0.33)	0.41	0.03	0.02	0.01
FLD030-2	(0.33)	>300	0.44	0.35	0.11
FLD031-1	(0.33)	>300	>300	>300	268.48
FLD032-1	(0.33)	>300	>300	>300	247.57
FLD035-3	(0.33)	14.11	>300	>300	257.05
FLD036-3	(0.33)	13.73	>300	>300	135.22
FLD037-1	(0.33)	>300	1.64	1.25	0.32
FLD060-1	(0.67)	>300	0.59	0.51	0.15
FLD060-2	(0.67)	>300	>300	>300	127.11
FLD061-1	(0.67)	>300	0.66	0.58	0.17
FLD061-2	(0.67)	>300	>300	>300	155.31
FLD064-1	(0.67)	>300	>300	>300	114.86
FLD067-1	(0.33)	>300	1.47	1.20	0.31
FLD067-3	(0.33)	20.08	186.68	150.37	40.97
FLD069-1	(0.33)	>300	>300	>300	125.96
FLD070-1	(0.33)	>300	2.52	0.68	0.18
FLD071-3	(0.33)	2.52	0.36	0.34	0.08
GEO026-3	(0.11)	2.15	20.34	19.16	2.35
GEO030-3	(0.44)	8.04	>300	271.72	30.90
GEO032-3	(0.25)	1.16	>300	292.07	32.02
GEO033-3	(0.38)	4.81	>300	>300	39.41
GEO041-3	(0.22)	0.21	42.28	32.90	3.60
GEO051-3	(0.25)	7.26	>300	>300	56.82
GEO064-3	(0.12)	0.33	>300	>300	55.07
GEO065-3	(0.12)	0.34	>300	>300	55.11
GEO066-3	(0.12)	0.32	>300	>300	55.14
GRP008-1	(0.22)	0.69	1.00	0.72	0.12
HEN007-6	(0.17)	0.12	>300	>300	211.72
LCL045-1	(0.20)	98.03	1.31	0.90	0.50
LCL097-1	(0.20)	0.26	0.67	0.20	0.12
LCL111-1	(0.20)	0.13	0.20	0.14	0.07
LCL130-1	(0.20)	0.01	0.03	0.01	0.02
LCL195-1	(0.20)	error	18.76	15.23	6.93
LCL230-1	(0.40)	error	209.09	133.76	61.13
LCL231-1	(0.40)	error	>300	189.14	85.31

Problem	Rating	Otter	leanCoP	lolliCoP	lolliCoP$_2$
NUM009-1	(0.12)	3.40	75.63	49.37	4.44
NUM283-1.005	(0.20)	0.43	0.28	0.20	0.17
NUM284-1.014	(0.20)	0.89	180.54	147.55	129.91
PLA004-1	(0.40)	>300	4.00	3.06	2.46
PLA004-2	(0.40)	>300	5.97	5.09	3.90
PLA005-1	(0.40)	>300	0.44	0.36	0.24
PLA005-2	(0.40)	>300	0.10	0.06	0.03
PLA007-1	(0.40)	>300	0.14	0.13	0.08
PLA008-1	(0.40)	>300	251.50	204.21	142.95
PLA009-1	(0.40)	>300	0.06	0.05	0.03
PLA009-2	(0.40)	>300	2.10	1.73	1.20
PLA010-1	(0.40)	>300	250.53	203.04	142.21
PLA011-1	(0.40)	>300	0.14	0.09	0.05
PLA011-2	(0.40)	>300	0.45	0.36	0.24
PLA012-1	(0.40)	>300	66.06	52.21	36.79
PLA013-1	(0.40)	>300	0.24	0.18	0.11
PLA014-1	(0.40)	>300	2.06	1.61	1.21
PLA014-2	(0.40)	>300	2.13	1.75	1.33
PLA016-1	(0.40)	>300	0.07	0.07	0.04
PLA019-1	(0.40)	>300	0.06	0.05	0.03
PLA021-1	(0.40)	>300	0.18	0.13	0.07
PLA022-1	(0.40)	>300	0.40	0.32	0.24
PLA022-2	(0.40)	>300	0.03	0.02	0.01
PLA023-1	(0.40)	>300	72.74	57.73	40.71
PUZ034-1.004	(0.67)	error	15.87	12.42	9.83
RNG006-2	(0.20)	4.69	0.26	0.35	0.06
RNG040-1	(0.11)	0.05	0.01	0.01	0.01
RNG040-2	(0.22)	0.10	0.21	0.19	0.04
RNG041-1	(0.22)	0.16	43.86	36.25	6.37
SET014-2	(0.33)	176.24	174.31	134.35	27.97
SET016-7	(0.12)	>300	10.99	8.29	1.05
SET018-7	(0.12)	>300	11.13	8.37	1.06
SET041-3	(0.44)	>300	59.88	45.36	4.88
SET060-6	(0.12)	0.19	0.04	0.03	0.00
SET060-7	(0.12)	0.33	0.05	0.03	0.00
SET083-7	(0.12)	24.39	40.34	34.70	5.41
SET085-6	(0.12)	12.72	>300	>300	65.58
SET085-7	(0.25)	65.79	46.01	33.55	5.22
SET119-7	(0.25)	177.97	60.35	48.50	6.71
SET120-7	(0.25)	181.62	60.23	48.46	6.71
SET121-7	(0.25)	178.42	72.81	55.77	7.63
SET122-7	(0.25)	180.13	72.83	55.82	7.64
SET152-6	(0.12)	0.45	3.50	2.60	0.38
SET153-6	(0.12)	>300	0.70	0.56	0.10
SET187-6	(0.38)	>300	18.01	13.53	2.27
SET196-6	(0.12)	10.59	>300	>300	196.13
SET197-6	(0.12)	10.63	>300	>300	196.06
SET199-6	(0.25)	>300	>300	>300	203.63
SET231-6	(0.12)	>300	12.86	9.74	1.63
SET234-6	(0.25)	>300	>300	>300	251.18
SET252-6	(0.25)	61.53	>300	>300	202.60
SET253-6	(0.25)	>300	>300	>300	203.24
SET451-6	(0.12)	>300	>300	>300	281.67
SET553-6	(0.25)	36.81	>300	>300	204.46
SYN048-1	(0.20)	0.00	0.00	0.00	0.00
SYN074-1	(0.11)	0.87	>300	>300	74.43
SYN075-1	(0.11)	0.17	>300	266.47	49.18
SYN102-1.007:007	(0.33)	1.00	39.38	39.70	22.95
SYN311-1	(0.20)	error	123.36	99.86	45.68

Table 3. Comparison of OTTER, leanCoP, and lolliCoP

(a) 33 problems solved by OTTER, leanCoP, and lolliCoP

	OTTER	leanCoP	lolliCoP	lolliCoP$_2$
Total CPU time	1143.03	1590.66	1139.41	338.47
Average CPU time	34.64	48.20	34.53	10.26
Speedup Ratio	1.39	1.00	1.40	4.70

(b) 36 problems solved by OTTER and lolliCoP

	OTTER	lolliCoP	lolliCoP$_2$
Total CPU time	1152.40	1969.67	450.57
Average CPU time	32.01	54.71	12.52
Speedup Ratio	1.71	1.00	4.37

(c) 76 problems solved by leanCoP and lolliCoP

	leanCoP	lolliCoP	lolliCoP$_2$
Total CPU time	2757.83	2038.58	853.24
Average CPU time	36.29	26.82	11.23
Speedup Ratio	1.00	1.35	3.23

In lolliCoP we already check whether each clause is ground or not at the time the clauses are added into the proof context in `pr/1`. We can further take advantage of that check by not only adding the clauses differently, but by adding different sorts of clauses. In lolliCoP a clause c (ground or not) is represented by the Lolli clause `cl(`c`)`. We can continue to represent ground clauses in the same way, but when c is non-ground, instead represent it by the Lolli clause: `cl(C1) :- copy_term(`c`,C1).` When this clause is used, it will return not the original clause, c, but a copy of it. To be precise, we replace the second clause of `pr/1` with a clause of the form:

```
pr([C|Mat]) :-
     (ground(C) -> (cl(C) -<> pr(Mat)
                 ; (forall C1\ cl(C1) :- copy_term(C,C1)) =>  pr(Mat)).
```

Note the use of explicit quantification over the variable `C1`.

In lolliCoP$_2$ the loaded clauses are further modified to take a second parameter, the path-depth limit. The Lolli clauses for ground clauses simply ignore this parameter. The ones for non-ground clauses check it first and proceed only if the limit has not yet been reached. In this version of the prover there is no check whatsoever for the ground status of a clause in the core (`prove/2`). This removes the potentially significant computational cost of checking the ground status each time a clause is selected: an operation linear in the size of the selected clause. Space constraints keep us from including the full program.

Taken together these small improvements actually triple the performance of the system. While the first optimization can be added, awkwardly, to leanCoP, it is not possible to do away entirely with the groundness check in that setting.

8 Conclusion

Lean theorem proving began with leanTAP [1], which provided an existence proof that it was possible to implement interesting theorem proving techniques using clear short Prolog programs. It was not expected, however, to provide particularly powerful systems. Recently, leanCoP showed that these programs can be at once perspicuous and powerful.

However, to the extent that these programs rely on the use of term-level Prolog data structures to maintain their proof contexts, they require the use of list manipulation predicates that are neither particularly fast nor clear. In this paper we have shown that by representing the proof context within the proof context of the meta-language, we can obtain a program that is at once clearer, simpler, and faster.

Source code for the examples in this paper, as well as the LLP compiler can be found at `http://www.cs.hmc.edu/~hodas/research/lollicop`.

References

1. B. Beckert and J. Posegga. leanTAP: lean tableau-based theorem proving. In *12th CADE*, pages 793–797. Springer-Verlag LNAI 814, 1994.
2. W. Bibel. *Deduction: Automated Logic.* Academic Press, 1993.
3. W. Bibel, S. Brüning, U. Egly, and T. Rath. KoMeT. In *12th CADE*, pages 783–787. Springer-Verlag LNAI 814, 1994.
4. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
5. James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, pages 391–405, Munich, Germany, 1996. Springer-Verlag LNCS 1101.
6. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstraction in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.
7. J. S. Hodas, K. Watkins, N. Tamura, and K.-S. Kang. Efficient implementation of a linear logic programming language. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 145–159, June 1998.
8. Argonne National Laboratory. Otter and MACE on TPTP v2.3.0. Web page at `http://www-unix.msc.anl.gov/AR/otter/tptp230.html`, May 2000.
9. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. Setheo: a high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.
10. W. MacCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
11. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
12. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. Setheo and E-Setheo—the CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246, 1997.
13. G. Nadathur and D. J. Mitchell. Teyjus—a compiler and abstract machine based implementation of lambda Prolog. In *6th CADE*, pages 287–291. Springer-Verlag LNCS 1632, 1999.
14. J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. In *Proceedings of the Third International Workshop on First-Order Theorem Proving*, pages 152–157. University of Koblenz, 2000. Electronically available, along with submitted journal-length version, at `http://www.intellektik.informatik.tu-darmstadt.de/~jeotten/leanCoP/`.
15. G. Sutcliffe and C. Suttner. The TPTP problem library—CNF release v1.2.1. *Journal of Automated Reasoning*, 21:177–203, 1998.