Equality Preprocessing in Connection Calculi

Benjamin. E. Oliver[1] and Jens Otten[2]

[1] University of Oslo, Oslo, Norway
benjamin.e.oliver@gmail.com
[2] University of Oslo, Oslo, Norway
jeotten@ifi.uio.no

Abstract

Equality is a fundamental concept in first-order reasoning, yet for connection based proof methods a notoriously challenging one to handle efficiently. While paramodulation is a popular technique for resolution and related calculi, there is no single practical successful solution for connection based approaches. We present an extensible system for equality preprocessing in connection calculi (EPICC) that can be used as a tool in reducing the search space of problems that contain equality. We specify a number of preprocessing rules, describe an implementation of these rules and compare it with existing approaches for dealing with equality in connection calculi.

1 Introduction

Equality is an essential concept when reasoning about everyday life. Knowing that London is the capital of the *United Kingdom* and that the *United Kingdom* and the *UK* are equal, i.e. they denote the same country, lets us deduce that London is the capital of the *UK*. Indeed, it is not surprising that the largest collection of problems for automated reasoning or, more specifically, automated theorem proving (ATP), the TPTP library [17], uses equality in the formalisation of a vast number of problems. Yet, for a concept so ubiquitous its automation is not straightforward.

In mathematics and first-order logic, equality is usually represented by "=" and the above statement could (in short) be encoded using the first-order formula

$$(a = b \wedge P(a)) \Rightarrow P(b)$$

In order to solve this problem, ATP systems have to incorporate techniques for equality. While paramodulation [15] is a successful technique for dealing with equality in the popular resolution method [16], the situation is more complicated for tableau or connection calculi [9, 4]. For example, rigid E-unification [8] is not decidable and its use practically infeasible due to its complexity [7]. A more restricted technique called bounded rigid E-unification has been implemented in the tableau prover ePrincess [2, 3], but cannot easily be extended to connection calculi.

So far, the most successful technique for dealing with equality in connection calculi, as also implemented in the leanCoP prover [10, 13], adds the equality axioms and then uses *restricted backtracking* [11] to limit the amount of redundancy caused by the equality axioms. But as observed in the yearly system competitions CASC, the relative performance of leanCoP compared to other provers on problems with equality is significantly lower than its relative performance on problems without equality [18].

In this paper we present a framework for preprocessing techniques in order to simplify problems containing equality. Even though the presented approach can be used in combination with any ATP procedure, we have implemented, tested and evaluated it in combination with the connection prover leanCoP. It is also tested against an implementation of the Modification Method [6], another well-known preprocessing technique to deal with equality in connection calculi.

In Section 2, we first present the details of the underlying matrix method. Section 3 introduces the preprocessing steps and rules. Section 4 gives details on how these preprocessing rules have been implemented and combined with leanCoP. In Section 5 this implementation is compared to the equality technique currently used in leanCoP and an implementation of the Modification Method. The paper concludes with a summary and brief outlook on further research in Section 6.

2 Preliminaries

In this section some basic concepts and notations are introduced, such as the matrix characterization and the standard equality axioms.

2.1 First-Order Logic and Matrix Characterization

The standard notation for first-order formulae is used. *Terms* (denoted by s,t,u,v) are built up from functions (denoted by f), constants (a,b,c) and variables (denoted by x,y,z). An *atomic formula* (denoted by A) is comprised of predicate symbols (denoted by P) and terms. A *(first-order) formula* (denoted by F) is built up from atomic formulae, the connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and the first-order quantifiers $\forall$ and $\exists$. A *literal* L has the form A or $\neg A$. Its *complement* $\overline{L}$ is A if L is of the form $\neg A$; otherwise $\overline{L}$ is $\neg L$.

A formula in (disjunctive) *clausal form* has the form $\exists x_1 \ldots \exists x_n (C_1 \vee \ldots \vee C_n)$, where each clause C_i is a conjunction of literals $L_1, \ldots, L_{m_i}$.[1] It is usually represented as a set of clauses $\{C_1, \ldots, C_n\}$, which is called a (clausal) *matrix* $\mathbf{M}$. Every formula F can be translated into a validity-preserving formula F' in clausal form.

Definition 2.1 (Matrix)**.** *A set of clauses is represented as a matrix. A* matrix $\mathbf{M}$ *of a formula consists of its clauses* $\{C_1, \ldots, C_n\}$*, in which each* clause *is a set of its literals* $\{L_1, \ldots, L_m\}$*.*

In the *graphical representation* of a matrix, its clauses are arranged horizontally, while the literals of each clause are arranged vertically (see Figure 1).

A *connection* $\{P(...), \neg P(...)\}$ is a set of two literals with the same predicate symbol, of which (exactly) one is negated. A *first-order* or *term substitution* σ is a mapping from the set of term variables to the set of terms. In $\sigma(L)$ and $\sigma(C)$ all term variables x in L and C are substituted by their image $\sigma(x)$. A connection $\{L_1, L_2\}$ with $\sigma(L_1) = \sigma(\overline{L_2})$ is called σ*-complementary*. A *path* through a matrix $\mathbf{M} = \{C_1, \ldots, C_n\}$ is a set of literals that contains one literal from each clause $C_i \in \mathbf{M}$, i.e. a set $\cup_{i=1}^{n} \{L_i'\}$ with $L_i' \in C_i$. The following *matrix characterization* [5] provides a simple criterion for the validity of a formula and is the basis of the connection method [4]; see also [1].

Theorem 2.1 (Matrix Characterization)**.** *A formula F and its matrix* $\mathbf{M}$ *are valid iff there exists (1) a multiplicity* $\mu : \mathbf{M} \to \mathbb{N}$ *(specifying the number of clause copies), (2) a term substitution* σ *and (3) a set of connections S, such that every path through the matrix* $\mathbf{M}^{\mu}$ *of F contains a* σ*-complementary connection* $\{L_1, L_2\} \in S$*. In* $\mathbf{M}^{\mu}$*, clause copies have been added according to* μ*.*

$$\left[\begin{bmatrix} L_1^1 \\ \vdots \\ L_i^1 \end{bmatrix} \quad \ldots \quad \begin{bmatrix} L_1^n \\ \vdots \\ L_j^n \end{bmatrix}\right]$$

Figure 1: Graphical representation of a matrix

[1] Even though the use of a *conjunctive clausal form* (cnf) is common, a disjunctive clausal form (dnf) is used for historical and practical reasons; the difference between both forms is marginal (a formula F in dnf is *valid* iff $\neg F$ in cnf is *unsatisfiable*).

2.2 Equality

In order to extend the language to first-order logic *with equality*, the (predefined) *equality predicate* $\approx$ is added. Instead of $\approx (s,t)$ we use the common infix notation $s \approx t$. We also use $s \not\approx t$ as an abbreviation for $\neg(s \approx t)$. One way to specify the interpretation (or meaning) of equality is by adding equality axioms. Once these axioms have been added the equality symbols $\approx$ and $\not\approx$ can be treated as uninterpreted predicates.

Definition 2.2 (Equality Axioms)**.** *We will use the notation* $\alpha(\mathbf{M})$ *to denote the set of axiom clauses that must be generated for the matrix* $\mathbf{M}$ *of a formula and* $\mathbf{M} \cup \alpha(\mathbf{M})$ *to indicate the resulting matrix formed by combining the original matrix and the axioms. If* $\mathbf{M}$ *does not contain equality then* $\alpha(\mathbf{M})$ *is an empty set, however if* $\mathbf{M}$ *contains equality, then* $\alpha(\mathbf{M})$ *is the least set such that:*

$$\left[\begin{bmatrix} x \not\approx x \end{bmatrix} \quad \begin{bmatrix} x \approx y \\ y \not\approx x \end{bmatrix} \quad \begin{bmatrix} x \approx y \\ y \approx z \\ x \not\approx z \end{bmatrix}\right] \subseteq \alpha(\mathbf{M}) \qquad \text{(reflexivity, symmetry, transitivity)}$$

$$\begin{bmatrix} x_1 \approx y_1 \\ \vdots \\ x_n \approx y_n \\ f(x_1 \cdots x_n) \not\approx f(y_1 \cdots y_n) \end{bmatrix} \in \alpha(\mathbf{M}) \qquad \text{(for every function } f \text{ of arity } n \text{ in } \mathbf{M})$$

$$\begin{bmatrix} x_1 \approx y_1 \\ \vdots \\ x_n \approx y_n \\ P(x_1 \cdots x_n) \\ \neg P(y_1 \cdots y_n) \end{bmatrix} \in \alpha(\mathbf{M}) \qquad \text{(for every predicate } P \text{ of arity } n \text{ in } \mathbf{M})$$

3 Equality Preprocessing

Unlike the Modification Method and its derivatives, the following approach is not designed to eliminate equality entirely. Instead, it is best viewed as a set of rules that aim to balance three properties - ease of implementation, good algorithmic complexity and, finally, improvement of the performance of the proof procedure. A combination that is difficult to achieve at the same time.

3.1 Basic Notation

3.1.1 Matrix Notation

As both matrices and clauses are really nothing but sets, we will employ a special notation that allows us to focus on certain properties. The notation $\mathbf{M} = [\mathbf{C}\ S]$ is a visual representation of $\mathbf{M} = \{\mathbf{C}\} \cup S$, in which $\mathbf{C}$ is a clause and S is a set of clauses. Note that it is perfectly fine for S to be empty. In the same way we can visualize $\mathbf{C} = \{s \approx t\} \cup \mathscr{C}$ by using vector column notation. By combining these we can pattern match against certain *literals* and sub-clauses/matrices that are of interest. For example

$$\mathbf{M} = \left[\begin{bmatrix} s \approx t \\ \mathscr{C} \end{bmatrix} \quad S\right]$$

lets us match a clause ($\mathbf{C}$) consisting of the literal $s \approx t$, the remaining (possibly empty) literal set $\mathscr{C}$ and the remaining (possibly empty) set of clauses S.

3.1.2 Most General Unifiers

Given a set of equations $E = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$, a unifier is a (variable) substitution σ such that $s_1\sigma = t_1\sigma, \ldots, s_n\sigma = t_n\sigma$. Given a set of unifiers $U(E)$ a substitution σ is a most general unifier (mgu) of E if $\forall\theta \in U(E) : \theta$ is an instance of σ.

3.1.3 Rules

The following preprocessing rules should be read top down. If the conditions presented above the bar hold then one can infer the result below. $\mathbf{M}_1 \vDash \mathbf{M}_2$ means that $\mathbf{M}_2$ is a logical consequence of $\mathbf{M}_1$, $\vDash \mathbf{M}$ means that $\mathbf{M}$ is valid.

3.2 Valid Clauses

Any matrix that contains a clause $\mathbf{C}$ consisting only of positive equalities is valid if there exists a most general unifier (mgu) for $\mathbf{C}$. Note that by this rule, an empty clause is valid.

$$\frac{\left[\mathbf{C} \quad S\right] \quad \exists\sigma : \sigma = mgu(\mathbf{C}) \quad \forall x \in \mathbf{C} : \; x = (s \approx t)}{\vDash [\mathbf{C}] \cup \alpha([\mathbf{C}])}$$

As all variables local to $\mathbf{C}$ are existentially quantified an mgu for such a clause represents an assignment for every variable in $\mathbf{C}$ such that every equality literal is true. If the mgu is empty then every element of $\mathbf{C}$ must have been of the form $s_1 \approx s_1, \ldots, s_n \approx s_n$. One can construct a new matrix $\mathbf{M}'$ consisting of the clause $\mathbf{C}$ extended with the axioms of equality such that the resulting matrix is valid. As $\mathbf{M}'$ is a subset of $\mathbf{M}$ then $\mathbf{M} \cup \alpha(\mathbf{M})$ is valid. This rule not only provides a termination case for the reduction algorithm, but allows certain (artificial) theorems to be proven in a very efficient manner.

In order to ensure that the reader is under no illusion, we will take some time to clarify this first rule. There are three conditions that must be met, read from left to right they are: The presence of a clause $\mathbf{C}$. The existence of a substitution σ such that σ is the MGU for clause $\mathbf{C}$ (this does not preclude the empty substitution $\sigma = \{\}$). Finally, if the clause $\mathbf{C}$ is not empty, then it must only contain positive equality literals. If these conditions are met, then we can construct a new matrix consisting of the single clause $\mathbf{C}$. If we union this matrix with the set of all axioms that $\mathbf{C}$ generates then the result can be shown to be valid using first-order logic alone.

As an example, consider the following: $\{\{x \approx a,\ f(x) \approx f(a)\}, \ldots\}$. While there may be multiple ways to prove the validity of such a formula - we can see that all we need to show is that there exists a value x that is equal to a. Due to the reflexivity of equality such a search is rather straightforward. Even though the occurrence of such a formula may be uncommon, detecting such a clause can be crucial for a successful proof search. The exclusion of this rule a can lead to a proof procedure timing out before *reaching* the clause in question.

3.3 Contradictions

When considering a formula that is equality free then any clause $\mathbf{C}$ that contains $P(s)$ and $\neg P(s)$ is contradictory and can be removed. This idea can be extended to clauses containing $P(s)$ and $\neg P(t)$ if the remaining clause $\mathscr{C}$ implies that s and t are equal. If we can derive $s \approx t$ in $\mathscr{C}$ then both $(P(s) \wedge \neg P(t) \wedge \mathscr{C})$

and $(s \not\approx t \wedge \mathscr{C})$ must result in contradictory clauses.

$$\frac{\left[\begin{bmatrix} P(s_1,\ldots s_n) \\ \neg P(t_1,\ldots t_n) \\ \mathscr{C} \end{bmatrix} \quad S\right] \quad \mathscr{C} \vDash s_1 \approx t_1 \wedge \cdots \wedge s_n \approx t_n}{\vDash S \cup \alpha(S) \quad \text{iff} \quad \vDash \mathbf{M} \cup \alpha(\mathbf{M})}$$

In the same way as for predicates, if we derive that $a \approx b$ in the remainder of the clause, then it can not be the case that both $a \approx b$ and $a \not\approx b$ can be true simultaneously.

$$\frac{\left[\begin{bmatrix} s \not\approx t \\ \mathscr{C} \end{bmatrix} \quad S\right] \quad \mathscr{C} \vDash s \approx t}{\vDash S \cup \alpha(S) \quad \text{iff} \quad \vDash \mathbf{M} \cup \alpha(\mathbf{M})}$$

3.4 Redundancy

Consider a clause that contains two equalities $a \approx b$ and $f(a) \approx f(b)$. If $a \approx b$ is true then $f(a) \approx f(b)$ follows, meaning that $f(a) \approx f(b)$ is redundant. However, $a \approx b$ does not follow from $f(a) \approx f(b)$. The same principle can be used to find *redundant* predicates.

$$\frac{\left[\begin{bmatrix} s \approx t \\ \mathscr{C} \end{bmatrix} \quad S\right] \quad \mathscr{C} \vDash s \approx t}{\vDash \begin{bmatrix} \mathscr{C} & S \end{bmatrix} \cup \alpha(\begin{bmatrix} \mathscr{C} & S \end{bmatrix}) \quad \text{iff} \quad \vDash \mathbf{M} \cup \alpha(\mathbf{M})}$$

The rule of redundancy is an interesting one. While it may seen advantageous to minimise the number of literals in a clause, we must remember that our aim is to reduce the search space of the problem. It may well be the case that an equation or predicate that is redundant in terms of derivability (one could in theory generate such a literal), is key to the proof of a formula. If this is the case then the responsibility is on the theorem prover to - in some sense - re-find this literal.

3.5 Pure Clauses

Assume that $P(s) \in \mathbf{C}_j$. If there does not exist a k such that $\neg P(t) \in \mathbf{C}_k$ then $\mathbf{C}_j$ is isolated; the same holds for $\neg P(s) \in \mathbf{C}_j$ if there is no $P(t)$. An isolated clause may be removed from the matrix. This general rule is applicable to formulae that contain equality.

3.6 Unsatisfiable Clauses

If a matrix without equality axioms does not contain negated equality then the only instance (of negated equality) can come from the addition of the equality axioms. A negated equality has to be part of the connection to make an appropriate path complementary. As every equality axiom apart from reflexivity contains at least one positive equality again, the only possible connection *to make this path complementary* has to be to the literal in the reflexivity axiom. This is equivalent to finding the mgu for the positve equation. If no mgu exists then the clause is contradictory and can be removed.

$$\frac{\left[\begin{bmatrix} s \approx t \\ \mathscr{C} \end{bmatrix} \quad S\right] \quad \exists \sigma : \sigma = mgu(s \approx t) \quad \forall (u,v) : (u \not\approx v \notin \mathbf{M})}{\vDash \begin{bmatrix} \sigma(\mathscr{C}) & S \end{bmatrix} \cup \alpha(\begin{bmatrix} \sigma(\mathscr{C}) & S \end{bmatrix}) \quad \text{iff} \quad \vDash \mathbf{M} \cup \alpha(\mathbf{M})}$$

$$\frac{\left[\begin{bmatrix} s \approx t \\ \mathscr{C} \end{bmatrix} \quad S\right] \quad \nexists \sigma : \sigma = mgu(s \approx t) \quad \forall (u,v) : (u \not\approx v \notin \mathbf{M})}{\vDash S \cup \alpha(S) \quad \text{iff} \quad \vDash \mathbf{M} \cup \alpha(\mathbf{M})}$$

Thus, if there exists a matrix $\mathbf{M}$ that contains positive equality but does not contain negated equality, then it can be reduced to a matrix $\mathbf{M}'$ that is equality free such that the transformation is sound and complete.

3.7 Unit Clause

The final rule that we will consider concerns negated unit clauses. Consider a formula of the form where s and t are variable free:

$$(s \approx t) \Rightarrow P(s) \vee \neg P(t)$$

When converted to disjunctive normal form this will result in a matrix that contains a negated unit clause:

$$\left[\left[s \not\approx t\right] \quad \left[P(s)\right] \quad \left[\neg P(t)\right]\right]$$

If the negated equality clause is used in a proof then every path must pass through it.

If $s \not\approx t$ is contradictory then the original formula was of the form $(s \approx s) \Rightarrow F$, in which case the matrix becomes F. If this is not the case then the following holds

$$\frac{\left[\left[s \not\approx t\right] \quad S\right] \quad vars(s \not\approx t) = \emptyset}{\vDash \mathbf{M} \cup \alpha(\mathbf{M}) \quad \text{if} \quad \vDash S\{s \mapsto t\} \cup \alpha(S\{s \mapsto t\})}$$

$$\frac{\left[\left[s \not\approx t\right] \quad S\right] \quad vars(s \not\approx t) = \emptyset}{\vDash \mathbf{M} \cup \alpha(\mathbf{M}) \quad \text{if} \quad \vDash S\{t \mapsto s\} \cup \alpha(S\{t \mapsto s\})}$$

where $vars(s \not\approx t)$ is the set of all variables in $s \not\approx t$ and $S\{t \mapsto s\}$ is the set (of remaining clauses) S in which all terms t have been replaced by the term s. This rule is sound as every path through $\mathbf{M}$ contains the negated equation $s \not\approx t$, and applications of the equality axioms can be used to replace any occurrence of s by t or vice versa. While soundness is preserved, completeness of the calculus is lost. While this may sound problematic, we will see that in "real world" situations such a limitation has little to no negative impact.

One special case is that when $s \not\approx s$ we know that $\vDash M \cup \alpha(\mathbf{M})$ iff $\vDash S \cup \alpha(S)$ by the rule of contradictions 3.3. Out of all of the rules discussed, this one is the most powerful. As we will see in the evaluation section, the ability to make *good* choices in terms of choosing a direction (or not applying the rule) is an important factor.

4 Implementation

The aim of this work is to improve the performance of equality handling for connection calculi – with a particular focus on the leanCoP theorem prover. As the rules discussed in Section 3 will be run on large matrices they needed to be performant.

Such an approach has certainly influenced both the design of the EPICC [2] [3] run-time system, and the considerations made when searching for rules to implement. The rules that we have seen in Section 3

[2] Available for download under the GPL license at http://leancop.de/epicc/.

[3] The Clojure implementation of the preprocessing steps that does not include the leanCoP core prover can be obtained at https://github.com/beoliver/clj-epicc.

were influenced not only by the nature of the connection calculus, but also the internal representation of the matrix and the fact that we plan on using it as a preprocessing step. The preprocessing rules of the previous section have been implemented in the EPICC (**E**quality **P**reprocessing **i**n **C**onnection **C**alculi) system.

The current version is written in the functional Lisp-like language Clojure that runs on the JVM (Java Virtual Machine). The data driven approach of the language makes it ideal for prototyping and exploration. The current rule based approach has been developed with portability in mind (an implementation in Haskell is being actively developed).

The current approach taken by the leanCoP theorem prover is as follows: If a matrix $\mathbf{M}$ contains an equality symbol then the matrix is extended with the axioms of equality before the actual proof search starts, proving the matrix $\mathbf{M} \cup \alpha(\mathbf{M})$. EPICC replaces this procedure by first applying the rules discussed in Section 3, before generating the set of axioms $\alpha(\mathbf{M})$. For the current implementation, formulae in a disjunctive normal form matrix format are accepted.

Internally, clauses are indexed sets of literals. A matrix is represented as a mapping from clause indexes to the clauses as well as additional information such as the indexes of clauses that contain positive/negated equality and a mapping from terms to the clauses that they occur in (an important factor when it comes to performance). These tables are updated every time a clause is added/removed from the matrix, meaning that when implementing a rule that, say, only considers clauses containing positive equality, one does not need to test every clause. The same principle is true when performing global rewriting – one only needs to update the clauses that a term occurs in. Reductions are expressed in terms of *rules*. The concepts of programming against interfaces and using well-defined return values are common features of many popular languages (both functional and imperative). A local rule is something that implements two functions.

```
(defprotocol LocalRule
  (candidate-clauses [rule matrix])
  (apply-local [rule matrix clause]))
```

The same idea can be expressed in Haskell using data types.

```
data Rule = Rule { candidate_clauses :: Matrix -> [Int]
                 , apply_local :: Matrix -> Clause -> Result }
```

The function `candidate-clauses` is responsible for returning a sequence of all of the indexes that the rule may be applicable to. It is assumed that the search for candidates is cheap. By separating a rule in this way we gain the ability to arbitrarily terminate the reduction process (either due to time constraints or having found a solution) while retaining the most current version of the matrix. Moreover, rules can be added, deleted, and re-ordered in a straightforward manner aiding the development of new approaches.

As a concrete example, let us consider the rule for valid clauses and how it could be implemented in the Clojure language (figure 2).

The local rules are most commonly responsible for *deciding* if a given clause is valid, redundant, or contradictory. Deletion and termination is handled by the reduction function. Variants exist that allow a rule to return a new clause, for example removing some $s \approx s$. In this case the reduction function will update the matrix to reflect the changes.

4.1 The Supervisor process

A *supervisor* is used that manages both state changes and termination conditions in order to run the rules. This *supervisor* is implemented using a function that continually loops (Clojure's "loop" keyword can

$$\frac{\left[\mathbf{C} \quad S\right] \quad \exists \sigma : \sigma = mgu(\mathbf{C}) \quad \forall x \in \mathbf{C} : \; x = (s \approx t)}{\vDash [\mathbf{C}] \cup \alpha([\mathbf{C}])}$$

```
(defrecord Valid_Clause []
  LocalRule
  (candidate-clauses [rule m]
    (filter #(only-pos-eq? (lookup m %)) (pos-eq-clauses m)))
  (apply-local-rule [rule m c]
    (cond (apply mgu? (formulae c))    (valid "...")
          (empty? (neg-eq-clauses m)) (redundant "...")
          :else                        no-op)))
```

Figure 2: An example of how the rule for valid clauses can be encoded in the Clojure language

.

be thought of as a recursive while loop that allows arguments to be passed) waiting for a termination condition. This means that a *local* rule is responsible for *deciding* if a given clause is valid, redundant or contradictory. Any deletion, termination or global re-writing is handled by the *supervisor* not the rule.

The *supervisor* tracks the state of the matrix as well as keeping a history of all rules that were applicable (i.e. they did not return NoOp). One result of this is that if a certain application of rules yields a valid result, then the system can extract the submatrix from the original input, add the axioms of equality and pass it on to a theorem prover. The rules are run until a terminating condition is met. Such a condition may either be a timeout, reaching a certain number of iterations, finding a solution or the previous iteration producing no change. The axioms of equality are added to the resulting matrix when no more reductions can be performed. It is possible to disable the equality axiom generation if desired.

Because it is possible for a rule to return "valid" or "invalid", the supervisor can be seen as a partial proof procedure. While this functionality is currently not used explicitly by EPICC it was noted that during testing, the procedure was able to prove the validity of a handful of problems from the TPTP library directly. In terms of current implementation, the resulting matrix is always passed to the leanCoP theorem prover. In the case of the rules alone deriving "valid", the resulting matrix can be proven by leanCoP .

While this does not affect the correctness of the testing results, it does mean that the EPICC framework requires user to know about which rules they are planning on using. One would imagine that in a future version of EPICC this information would be handled by the supervisor. Not only would this allow for a minor optimization in not having to invoke a theorem prover in all cases, but it would make the system more user friendly.

4.2 Internal representations and data structures

Internally clauses are represented as sets of literals. A matrix is represented as a mapping from clause indexes to the relevant set of literals (clause). An index represents the position of the clause in input matrix.

When the matrix is imported, addition information is gathered, such as indexes of clauses that contain positive/negated equality. An internal table is also built up mapping terms to the clauses that they occur in. The reason for this table is to improve performance when performing term substitutions – by knowing which clauses contain a term s, the cost of applying the substitution $\{s \to t\}$ is reduced to the number of clauses that currently contain the term s (as opposed to naïvely applying the substitution to

every clause). These tables are updated every time a clause is added/removed from the matrix, meaning that when implementing a rule that, say, only considers clauses containing positive equality, one does not need to repeatedly check every clause in the input matrix.

5 Evaluation

The equality preprocessing techniques described in Section 3 and implemented in Section 4 were evaluated on all 8044 FOF problems contained in the TPTP library v6.4.0.

If a problem did not contain equality, or it caused a parser error then it was ignored. This resulted in a set of 4672 problems that would be used for testing. For every problem we have verified that the results are consistent with the TPTP status of the problem. Of the 4672 problems that the tests were run on, we are particularly interested in the 4189 that have the TPTP status of *Theorem*.

5.1 Method

Six transformation strategies were compared against each other. The Clojure implementation of the EPICC system was used to perform all strategies. The first of these strategies simply adds the axioms of equality. This is the current approach taken by leanCoP and as such the results would provide a baseline for the remaining five approaches. The second approach performed the Modification Method (STE). As this is a well known approach for handling equality that can be performed during preprocessing it is interesting to see how it compares to the EPICC techniques described in this paper. The third approach would not perform any translation, nor would it add the axioms of equality. The remaining three configurations were variations of the EPICC techniques. Such a selection provides us with the ability to both compare the transformations discussed with the current leanCoP technique and to compare the transformations with two more approaches.

Each of the six methods were evaluated on all 8044 FOF problems contained in the TPTP library v6.4.0. TPTP is a library of test problems which supports the testing and evaluation of ATP systems. The library is divided into problem "domains". Some examples of the domains are software verification (SWV and SWW), where it is formally established that a computer program does the task it is designed for, software creation (SWC), which is used to form a computer program that meets given specifications, category theory (CAT), general algebra (ALG), graph theory (GRA), and managment (MGT), the study of systems, and their use and production of resources.

As opposed to reading problems directly from the TPTP library, leanCoP was used to first convert each of the files into disjunctive normal form. If a formula did not contain equality then it would not be used in the benchmarking tests. The resulting matrices were saved to disk and individually read by EPICC.

Every problem in the TPTP library has a status. We will concern ourselves with "Theorem" and "Non Theorem". For every problem it was recorded whether the result produced by a method is coherent with the TPTP status of the problem. This allows us to see which methods result in errors due to the method not preserving the completeness of the input formula. Such an approach also allows us to verify that no method implementation results in an unsound theorem prover. The six approaches to be used were:

- Axioms **AX**. These results are used To provide a set of baseline results that other methods can be compared against. The input file is to be read by EPICC and the axioms of equality added before passing the resulting matrix to leanCoP. This corresponds to the technique the full leanCoP prover uses.

- The Modification Method **MM**. The input is read and Brand's Modification Method performed on the matrix before passing it to leanCoP. As we have seen, the Modification Method is an existing preprocessing technique that can be used to *eliminate* equality. An implementation was written that would accept matrices in disjunctive normal form. It should be noted that this implementation is naïve in the sense that no attempt was made to optimize it. The algorithm is based on the one outlined in Brands paper [6].

- No axioms **NO-AX**. The file is read by EPICC and the matrix passed to leanCoP without adding any axioms of equality. The reason for including this approach is to provide an insight into the practical need for explicit equality handling. If a formula can be shown to be a theorem without adding the axioms of equality (i.e. without interpretation of equality) then the clauses containing equality were redundant. This approach cannot guarantee to preserve the completeness for formulae containing equality.

- **EPICC-1**. A configuration of EPICC that preserves the completeness for formulae containing equality. This is achieved by not applying the *NegatedUnitClause* rule. The axioms of equality are added after no more transformations can be applied. The following (complete) rules are used: *EmptyClause*, *PositiveEqualityMGU*, *ContradictoryPredicates*, *ContradictoryNegations*, *IsolatedPredicates*.

- **EPICC-2**. A configuration of EPICC that uses a left-to-right rewrite rule for *NegatedUnitClause*. The axioms of equality are added after no more transformations can be applied. The following (complete) rules were used: *EmptyClause*, *PositiveEqualityMGU*, *ContradictoryPredicates*, *ContradictoryNegations*, *IsolatedPredicates* extended with a *NegatedUnitClause* rule which performs rewriting in a left-to-right manner meaning that $(s \not\approx t)$ would result in a global substitution $\sigma = \{s \mapsto t\}$.

- **EPICC-3**. A configuration of EPICC that uses a custom rewrite rule for *NegatedUnitClause*. The axioms of equality are added after no more transformations can be applied. The difference between this approach and that of **EPICC-2** is how the rule *NegatedUnitClause* decides if a clause would result in the substitution $\sigma = \{s \mapsto t\}$, the substitution $\sigma = \{t \mapsto s\}$, or if the unit clause should be left alone. For example, one of the conditions is that a substitution $\sigma = \{s \mapsto t\}$ is only allowed if s does not occur in t.

For every input file (matrix) six different outputs were produced - each corresponding to one of the strategies listed above. The core (Prolog) prover of leanCoP 2.1 using its complete *core* strategy `[cut,comp(7)]` was then to be invoked for each output and allowed to run for at most ten seconds before being cancelled. The used core strategy uses *restricted backtracking*, which is switched off when a proof search depth of seven is reached [11]. If a timeout occurs, then "timeout" was recorded. The core prover does not add any axioms of equality. SWI-Prolog version 8.0.2 was used for running leanCoP.

The choice to avoid using the strategy scheduling features of leanCoP was made for two reasons. As the strategy `[cut,comp(7)]` is complete we know that if running leanCoP on the output of one of the six configurations being tested results in "Non Theorem" and the TPTP library has it marked with the status "Theorem", then that configuration did not preserve the completeness of the input formula. If we were to use strategy scheduling then leanCoP would ignore the result "Non Theorem" if the internal strategy that proved the result was not complete (for example `[scut]`). While we would eventually achieve the same result due to the last strategy that leanCoP employs (`[def]`) being complete, we might run out of time before this occurred. Secondly, we are interested in the effect that the transformations themselves have on the performance. It is assumed that if a particular leanCoP strategy is effective then that improvement would be seen across all transformations. Such a decision is certainly

open to debate. It may well be the case that a particular strategy of leanCoP amplifies the effect of a particular transformation. However, as we wish to generalise the implementation of the theorem prover, such an event is beyond the scope of this work.

All evaluations were conducted on a six-core 2.2 GHz Intel Core i7 Macbook Pro with 16 GB of RAM running MacOS 10.14.4. Each input file was parsed by the Clojure implementation of the EPICC system and any transformations performed before invoking leanCoP on the output. EPICC was run using Clojure 1.10.0 and Java 1.8.0_181.

When calculating the amount of time that a proof procedure takes, the results do not take into account the amount of time that was spent performing the transformation.

5.2 Experimental Results

Of the 8044 FOF problems contained in the tptp library 4672 problems contain equality. Of the 4672 problems containing equality that the tests were run on, we are particularly interested in the 4189 problems that have the tptp status of "Theorem". We will consider all results with respect to these 4189 problems. The reason for only considering the results of formulae that have the TPTP status of "Theorem" is because three of the methods tested (**NO-AX**, **EPICC-2** and **EPICC-3**) are known to not preserve the completeness of the input formula in the general case. In the case of the transformation **NO-AX** it is simply because it does not include the axioms of equality. For the other two it is due to their use of the *NegatedUnitClause* rule. Thus, if we get the result "Non Theorem" when using these methods, we do not know if it was derived because the original formula in the tptp library is a non theorem, or if it was due to the method of transformation.

Table 1 provides an overview of the results. Table 2 presents the results aggregated by problem difficulty, while table 3 presents the results broken down by domain. The three methods that use techniques described in this paper (**EPICC-1**, **EPICC-2** and **EPICC-3**) outperform all other methods including the standard leanCoP approach (**AX**) in the number of theorems proven under ten seconds. Of all the methods tested, the Modification Method (**MM**) has the worst performance. While no optimizations were made to this method, the fact that the standard approach of **AX** (that also includes no optimization) resulted in 795 more proofs certainly suggests leanCoP handles the increase of search space introduced by the axioms of **AX** better than the increase of search space due to the large number of new clauses introduced by **MM**.

Of the three EPICC approaches, the performance of the complete method **EPICC-1** is only marginally better than **AX**. This suggests that (for the problems considered) the re-writing rule *NegatedUnitClause* has a major impact. It would seem that either the (complete) methods described in Section 3 do little to reduce search space, or that the conditions required for their application are unfortunately not met frequently. **EPICC-2** performs better than **EPICC-1**. **EPICC-2** uses a left-to-right rewrite strategy for the *NegatedUnitClause* rule. The method that leads to the most proofs being found is **EPICC-3**. **EPICC-3** uses a slightly less aggressive rewrite strategy for the *NegatedUnitClause* compared to **EPICC-2**.

5.2.1 New Proofs

Figure 3 presents a visualization of the the 200 proofs found that could not be found using the standard **AX** approach. This graphic can be though of as an alternative to a Venn diagram. Multiple dots underneath a bar indicate that those solutions were found by multiple methods. The *intersection* row shows the cardinalilty of the intersections. The *solutions* column indicates the number of solutions that a particular method found that the standard **AX** approach could not. For example - of the 166 solutions that **EPICC-3** found, 97 were found by exactly one other method, namely **EPICC-2**.

	AX	MM	NO-AX	EPICC-1	EPICC-2	EPICC-3
proved	969	174	725	973	1002	1099
0 to 1sec.	814	116	629	814	841	922
1 to 10sec.	154	58	96	159	161	177
not proven by **AX**	-	11	31	9	109	166

Table 1: Summary of different equality handling techniques for the 4189 considered problems

		AX	MM	NO-AX	EPICC-1	EPICC-2	EPICC-3
rating	total						
0.0 to 0.09	706	496	144	373	498	476	**524**
0.1 to 0.19	441	221	25	147	219	238	**259**
0.2 to 0.29	388	96	4	73	98	112	**124**
0.3 to 0.39	351	73	0	68	74	88	**94**
0.4 to 0.49	374	46	0	32	47	49	**52**
0.5 to 0.59	360	22	0	16	22	23	**30**
0.6 to 0.69	275	8	0	7	8	**9**	**9**
0.7 to 0.79	297	6	0	**8**	6	6	6
0.8 to 0.89	324	**1**	0	**1**	**1**	**1**	**1**
0.9 to 0.99	247	0	0	0	0	0	0
1.0	426	0	0	0	0	0	0
total	4189	969	173	725	973	1002	**1099**

Table 2: All theorems proved under 10 seconds aggregated by problem rating. A **bold** font denotes the best performing method(s).

We can see that not only does **EPICC-3** find the most (166), but that many of the alternative methods appear to be subsumed by it. Indeed **EPICC-3** found 54 solutions that no other method could. **EPICC-2** only managed to find a single (unique) solution that **EPICC-3** (or any other method) could not. **EPICC-1** found no unique solutions.

The Modification method **MM** returned 6 unique solutions which is quite interesting considering that it only found 11 solutions in total that the axiomatic approach could not. Indeed its general performance was poor managing to find only 174 proofs (roughly 16% of the total achieved by **EPICC-3**) and never managing to find a solution for a problem with a rating higher than the 0.2 bracket. However, it cannot be ignored that new proofs were found that no other method could produce.

Perhaps the most striking of all the results are those of the **NO-AX** approach which managed to prove a total of 31 theorems that the standard approach could not, with 26 of those being unique to this method. Such a high percentage (84%) of unique solutions is not that surprising if we consider the fact that by not adding axioms the search space is drastically reduced. The fact that **NO-AX** returned 508 false negatives (Table 4) supports this assumption. Indeed it is worthy of note that the **NO-AX** approach proved two more theorems within the 0.7 rating range than any other method, namely SEU205+1 (0.77) and SEU241+2 (0.73).

We see that for the most part the proofs of **EPICC-3** are a superset of the proofs of **EPICC-2** and **EPICC-1**. Thus a combination of **EPICC-3**, **NO-AX** and **MM** would find 199 of the 200.

		AX	MM	NO-AX	EPICC-1	EPICC-2	EPICC-3
Domain	total						
AGT	52	**17**	9	**17**	**17**	**17**	**17**
ALG	148	**25**	2	12	**25**	18	**25**
CAT	13	**2**	0	**2**	**2**	**2**	**2**
COM	24	**6**	2	5	**6**	**6**	**6**
CSR	28	**3**	1	1	**3**	2	**3**
GEO	114	9	1	6	9	**10**	**10**
GRA	18	3	0	3	**4**	3	**4**
GRP	79	**6**	0	4	**6**	5	5
HAL	6	**1**	0	**1**	**1**	**1**	**1**
KLE	224	**16**	4	3	**16**	14	**16**
KRS	14	9	0	**10**	**10**	**10**	**10**
LAT	105	14	6	13	**16**	**16**	**16**
MGT	45	16	6	13	16	14	**17**
MSC	1	**1**	**1**	0	**1**	**1**	**1**
NLP	15	1	0	**2**	1	1	1
NUM	558	155	23	122	152	145	**158**
PRO	63	9	0	**11**	9	9	9
PUZ	10	**3**	**3**	0	**3**	1	**3**
REL	108	**1**	0	0	**1**	0	**1**
RNG	152	32	5	24	34	29	**39**
SCT	79	**6**	0	**6**	**6**	**6**	**6**
SET	441	194	37	147	194	179	**197**
SEU	750	195	22	131	195	183	**196**
SWB	113	**14**	5	8	**14**	**14**	**14**
SWC	422	32	11	1	32	116	**136**
SWV	318	154	30	148	155	155	**161**
SWW	248	**33**	1	31	**33**	**33**	**33**
SYN	14	**6**	4	0	**6**	**6**	**6**
TOP	27	**6**	0	4	**6**	**6**	**6**
total	4189	969	173	725	973	1002	**1099**

Table 3: All theorems proved under 10 seconds aggregated by problem domain. A **bold** font denotes the best performing method(s).

	AX	MM	NO-AX	EPICC-1	EPICC-2	EPICC-3
false negatives	0	0	508	0	2	0
% of results	0%	0%	12%	0%	0.05%	0%

Table 4: Evaluation of false negatives returned for the 4189 considered problems. A false negative occurs when an input theorem is transformed into an invalid formula.

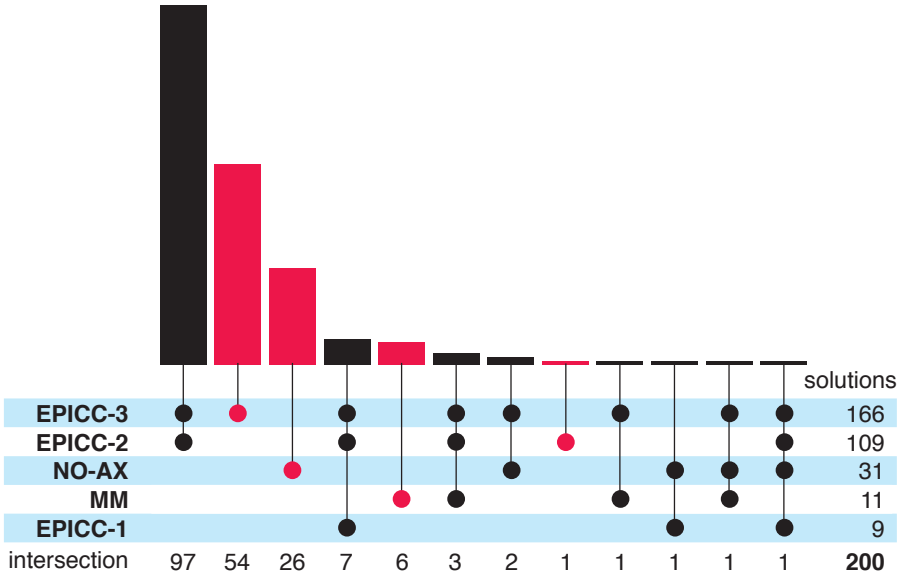

Figure 3: Visualising the "uniqueness" of the 200 proofs that could not be found using the standard leanCoP approach (**AX**).

6 Conclusion

The present paper introduces EPICC, a preprocessing technique for dealing with equality when proving formulae in (classical) first-order logic. Even though this technique can be used with any proof search calculus, it is in particular useful for tableau or connection calculi as integration of equality into these calculi is not straightforward.

The preprocessing technique has been specified using a set of rules for simplifying or modifying a matrix representing the original formula in clausal form. The rules have been implemented and tested with the connection prover leanCoP. This EPICC approach was compared to the modification method and the standard approach of adding the equality axioms to the original matrix.

Using the EPICC approach leanCoP was able to prove significantly more problems of the TPTP library than using its standard technique of just adding the equality axioms. This is in particular true for problems from the "Software Creation" (SWC) domain of the TPTP library (Figure 4). An interesting, yet to our knowledge so far undocumented, fact is that many of the problems in the TPTP library containing equality can be proved without any equality handling, i.e. treating the equality symbol as uninterpreted predicate symbol. The Modification Method proves significantly less problems than all other approaches. While the performance of the Modification Method may look undesirable, there were a handful of instances when it was the only approach that yielded a solution.

Future research work includes extending and optimizing the existent preprocessing rules. Furthermore, the adaptation and integration of similar preprocessing techniques into the non-clausal connection prover nanoCoP [12] or the non-classical provers ileanCoP and MleanCoP for first-order intuitionistic and modal logic [14] is currently investigated.

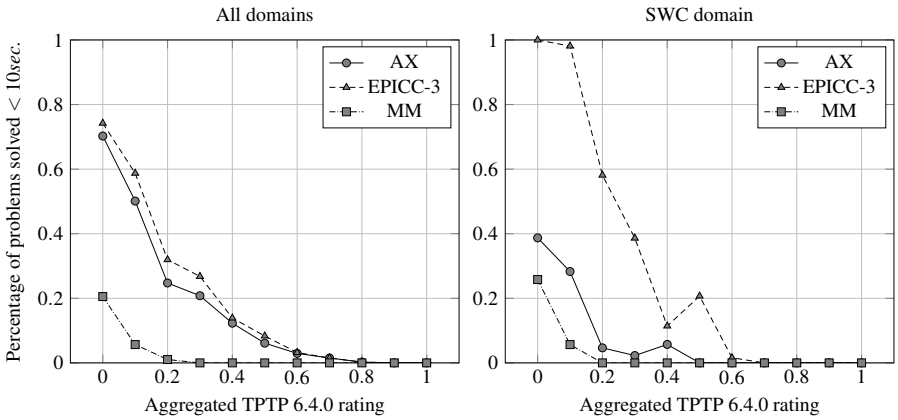

Figure 4: Percentage of theorems proven w.r.t. problem rating

Acknowledgements

The authors would like to thank the reviewers of a previous version of this paper for their comments.

References

[1] P. B. Andrews. Refutations by matings. *IEEE Transactions on Computers*, C-25(8):801–807, Aug 1976.

[2] Peter Backeman and Philipp Rümmer. Efficient algorithms for bounded rigid E-unification. In Hans De Nivelle, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 70–85, Heidelberg, 2015. Springer.

[3] Peter Backeman and Philipp Rümmer. Theorem proving with bounded rigid E-unification. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction – CADE-25*, pages 572–587, Heidelberg, 2015. Springer.

[4] Wolfgang Bibel. Matings in matrices. *Commun. ACM*, 26(11):844–852, 1983.

[5] Wolfgang Bibel. *Automated theorem proving*. Artificial intelligence. F. Vieweg und Sohn, Wiesbaden, 2nd edition, 1987.

[6] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[7] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid E-unification is undecidable. In Hans Kleine Büning, editor, *Computer Science Logic*, pages 178–190, Heidelberg, 1996. Springer.

[8] Jean Gallier, Paliath Narendran, Stan Raatz, and Wayne Snyder. Theorem proving using equational matings and rigid E-unification. *J. ACM*, 39(2):377–430, April 1992.

[9] Reiner Hähnle. Tableaux and related methods. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 101–178. Elsevier Science, Amsterdam, 2001.

[10] Jens Otten. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 283–291, Heidelberg, 2008. Springer.

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

[12] Jens Otten. nanoCoP: A non-clausal connection prover. In Nicola Olivetti and Ashish Tiwari, editors, *IJCAR 2016*, volume 9706 of *LNAI*, pages 302–312, Heidelberg, 2016. Springer.

[13] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1–2):139–161, 2003.

[14] Jens Otten and Wolfgang Bibel. Advances in connection-based automated theorem proving. In J. Bowen, M. Hinchey, and E.-R. Olderog, editors, *Provably Correct Systems*, Heidelberg, 2017. Springer.

[15] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 298–313. Springer, Heidelberg, 1983.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12:23–41, 1965.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[18] G. Sutcliffe. The 9th IJCAR Automated Theorem Proving System Competition - CASC-29. *AI Communications*, 31(6):495–507, 2018.