Congruence Classes with Logic Variables

JOE HURD, *Computer Laboratory, University of Cambridge, UK.*
E-mail: joe.hurd@cl.cam.ac.uk

Abstract

We are improving equality reasoning in automatic theorem-provers, and congruence classes provide an efficient storage mechanism for terms, as well as the congruence closure decision procedure. We describe the technical steps involved in integrating logic variables with congruence classes, and present an algorithm that can be proved to find all matches between classes (modulo certain equalities). An application of this algorithm makes possible a *percolation* algorithm for undirected rewriting in minimal space; this is described and an implementation in `hol98` is examined in some detail. [1]

Keywords: Congruence Closure, Equality Reasoning

1 Introduction

There has been a long-standing difficulty in theorem-proving of blending together deduction steps (such as Modus-Ponens or specialization) with equality steps (Leibniz' rule of substituting equals for equals). On the equality side, there are useful rewriters in most interactive theorem-provers: these are able to call decision procedures and perform 'simplifying' deductive steps. On the deduction side, there are powerful resolution theorem-provers such as Gandalf [14]: these include special deductive rules for handling equality but still this is where they are weak in practice.

We are interested in combining these two worlds, and in this paper we present a method to improve the handling of equality in deductive provers, using congruence classes. To motivate this work, consider the following example from group theory that a deductive prover would find difficult to handle:

$$\begin{array}{l} \{\forall\, x\, y\, z.\ [(x * y) * z = x * (y * z)] \wedge [e * x = x] \wedge [i(x) * x = e]\} \\ \Rightarrow \quad \{\forall\, x.\ x * i(x) = e\} \end{array}$$

Here the antecedent is enough to axiomatize the group operation $(*)$, identity (e) and inverse (i), and the consequent is the right inverse group law. Here is a proof in the form of a chain of equalities [3]:

$$\begin{array}{rcl} e & = & i(x * i(x)) * (x * i(x)) \\ & = & i(x * i(x)) * (x * (e * i(x))) \\ & = & i(x * i(x)) * (x * ((i(x) * x) * i(x))) \\ & = & i(x * i(x)) * ((x * (i(x) * x)) * i(x)) \\ & = & i(x * i(x)) * (((x * i(x)) * x) * i(x)) \end{array}$$

[1] Joe Hurd was supported by an EPSRC studentship

 ©Oxford University Press

$$\begin{aligned}
&= i(x*i(x))*((x*i(x))*(x*i(x))) \\
&= (i(x*i(x))*(x*i(x)))*(x*i(x)) \\
&= e*(x*i(x)) \\
&= x*i(x)
\end{aligned}$$

The problem is that the chain is long, and to find the proof a deductive prover would have to generate a very large number of terms using the given equalities. Note that this example would also pose a problem for a basic rewriter because the rewrite set requires completion to solve the problem.

Using congruence classes to store a set of terms maximizes subterm sharing and performs the congruence closure decision procedure, thus reducing the memory and time requirements of the deductive prover. There is however an incompatibility that needs addressing: deductive proving relies on logic variables that can be instantiated to arbitrary terms during the proof search, while congruence classes treat all variables in the terms as constants (and hence congruence closure does also).

The contribution made in this paper is an exploration of some of the ramifications of using congruence classes to store terms with logic variables, and a demonstration of how the underlying data structure may be exploited to find all possible term matches modulo certain equalities. An immediate application of this is a *percolation* algorithm that performs undirected rewriting on congruence classes; we also describe this and characterize what it can and cannot prove. As a secondary contribution, the algorithms presented in this paper have all been implemented in the `hol98` theorem-prover, and some preliminary results with the percolation algorithm are examined.

The structure of the paper is as follows: Section 2 defines a set of congruence classes with the basic operations, Section 3 is on matching: we precisely define the problem, present the match-finding algorithm and prove that it always terminates with all matches. We describe the percolation algorithm in Section 4, and in Section 5 we briefly examine the functionality and performance of the HOL implementation. We draw some conclusions from the work in Section 6, and finally in Section 7 relate our approach to others. Appendix A contains the details of the HOL implementation.

1.1 Notation

Terms in HOL are either constants, variables, function applications $app(t_1, t_2)$ of terms to terms, or lambda abstractions $lambda(v, t)$ of variables from terms.

2 Congruence Classes

2.1 Introduction

A congruence class set is a data structure that allows a set of terms to maximize the sharing of their subterms (and hence stores them in the minimum possible space). This property provides the basis of the congruence closure decision procedure [1] for the theory $\mathcal{E}$ in which the only interpreted function symbols are $=$ and $\neq$.

We define a **congruence class** to be a set of terms and a representative, so we can write a set of classes like this:

$$
\begin{array}{llllllll}
\{ & rep_1 & \{ & elt_{(1,1)}, & elt_{(1,2)}, & \ldots, & elt_{(1,m_1)} & \}, \\
 & rep_2 & \{ & elt_{(2,1)}, & elt_{(2,2)}, & \ldots, & elt_{(2,m_2)} & \}, \\
 & \ldots & & & & & & \\
 & rep_n & \{ & elt_{(n,1)}, & elt_{(n,2)}, & \ldots, & elt_{(n,m_n)} & \} \quad \}
\end{array}
$$

This class set $\mathcal{C}$ has n classes; we write C_i for the ith class, which here has m_i elements. The set of elements of a class can never be empty; the representative is always an element. We also insist that there is a distinguished **true class** containing $\top$ and a **false class** containing $\bot$. One more piece of notation: given an element elt in class C with representative rep, we will sometimes write $class(elt)$ for C and $[elt]$ for rep.

2.2 Normal Form

A term t is in **normal form** with respect to a congruence class set $\mathcal{C}$ if t is an element of a class in $\mathcal{C}$, and all the proper subterms of t are representatives of classes in $\mathcal{C}$. A congruence class set is in **normal form** if all its elements are in normal form.

In practice it is convenient to keep the class set $\mathcal{C}$ in normal form at all times. This is not a burden; the empty class set is already in normal form, and we add an arbitrary term t to $\mathcal{C}$ by the following procedure:

1. We may assume by recursion on the term structure that the immediate subterms of t are equal to terms in normal form.
2. Thus since the immediate subterms are equal to elements of $\mathcal{C}$ we may rewrite them to be their class representatives, to get t'.
3. If t' is an element of $\mathcal{C}$ then we are done, otherwise we add the new class $t'\{t'\}$.

This procedure adds t to the class set $\mathcal{C}$, and as a side effect produces the theorem $t = t'$ where t' is in normal form.

Henceforth we will assume that congruence class sets are always in normal form, unless we explicitly state otherwise.

2.3 Congruence Closure

We define a **close** operation on a class set $\mathcal{C}$: whenever two classes C_i and C_j in $\mathcal{C}$ have an element in common, then merge the two classes using the following algorithm:

1. First choose rep to be the smaller[2] term of rep_i and rep_j.[3]
2. Replace C_i and C_j in $\mathcal{C}$ with a new class C with representative rep and which contains all of the elements of C_i and C_j.
3. Finally we must rewrite all occurrences (in proper subterms) of rep_i and rep_j within $\mathcal{C}$ to rep.

So far the effect of this operation is the same as the standard congruence closure decision procedure, but we also need two more features in our close operation:

[2] Smaller here means fewer nodes in the term tree.

[3] Unfortunately, this is not quite enough to make our matching algorithm complete. In addition, for every logic variable X in rep that is not in both rep_i and rep_j, we must instantiate X in rep to an arbitrary value. This will always be possible, since if X is in rep_i but not rep_j, say, since the classes contain a common element we must have that $\forall\, x.\ rep_i[x/X] = rep_j$, so $rep_i[arb/X] = rep_j$ and $\forall\, x.\ rep_i[x/X] = rep_i[arb/X]$.

- Look for terms of the form $A = B$ in the true class, where A is not the same term as B, and merge the classes with representatives A and B.
- Look for terms of the form $A = A$ in all classes except the true class, and merge all the classes where we find such terms with the true class.

We now present a small example of congruence closure in action:[4]

1. Start with a minimal set of classes:
$$\begin{array}{llll} \{ & \top & \{ \ \top \ \}, & \\ & \bot & \{ \ \bot \ \} & \} \end{array}$$
2. Add the fact $f(f(f(a))) = a$ to the true class:
$$\begin{array}{lllll} \{ & \top & \{ \ \top, & a = a & \}, \\ & \bot & \{ \ \bot & & \}, \\ & f & \{ \ f & & \}, \\ & a & \{ \ a, & f(f(f(a))) & \}, \\ & f(a) & \{ \ f(a) & & \}, \\ & f(f(a)) & \{ \ f(f(a)) & & \} \quad \} \end{array}$$
3. Now when we add the fact $f(f(f(f(f(a))))) = a$, we get a neat collapse:
$$\begin{array}{lllll} \{ & \top & \{ \ \top, & a = a & \}, \\ & \bot & \{ \ \bot & & \}, \\ & f & \{ \ f & & \}, \\ & a & \{ \ a, & f(a) & \} \quad \} \end{array}$$
This happens because from $f(f(f(f(f(a))))) = a$ and $f(f(f(a)))$ we can deduce that $f(f(a)) = a$, and then this and $f(f(f(a))) = a$ together imply that $f(a) = a$.

There is nothing novel so far; the congruence closure decision procedure has already been implemented in HOL by Boulton [5] using term graphs, as part of his implementation of the Nelson-Oppen combination of decision procedures [12, 13]. We preferred the current data structure because it seemed easier to integrate logic variables and perform our matching algorithm on HOL terms, where we could use the standard HOL functions for operating on them. However, the underlying closure algorithm (using Union/Find) is the same in both cases, and both implementations perform a fully expansive proof.[5].

2.4 Terms with Logic Variables

We assume that our terms contain logic variables from a set V. To avoid confusion between the two types of variable, we will always write logic variables in upper case and normal 'HOL' variables in lower case. Thus the term $f(X)$ contains a logic variable but the term $f(x)$ does not.

We store terms in a congruence class set treating logic variables as normal variables. This has the effect that t_1 and t_2 will be in the same congruence class if and only if for every variable instantiation σ we have $\sigma t_1 = \sigma t_2$.

We are now in a position to define **equality modulo** $\mathcal{C}$ for a class set $\mathcal{C}$, which we write $=_\mathcal{C}$. If $t_1 =_\mathcal{C} t_2$, then t_1 can be transformed to t_2 by a sequence of rewriting

[4] The same example is also in the appendix with more implementational details.

[5] A fully expansive proof is one that completely reduces to the primitive axioms and rules of inferences of the logic

operations which always replace an element of $\mathcal{C}$ with another element in the same class, *treating logic variables as constants.*

To illustrate this central definition, if we have a class in $\mathcal{C}$ containing both X and $X+0$, then we can certainly say that $X =_{\mathcal{C}} X + 0$ and $7 + ((X + 0) + 0) =_{\mathcal{C}} 7 + X$, but not that $5 + 0 =_{\mathcal{C}} 5$ or that $Y + 0 =_{\mathcal{C}} Y$.

We also define $t \in_{\mathcal{C}} C_i$ to mean $t =_{\mathcal{C}} rep_i$.

Note that $=_{\mathcal{C}}$ is an equivalence relation, and if $\mathcal{C}$ is closed then for all t there will be at most one C_i such that $t \in_{\mathcal{C}} C_i$.

3 Matching Algorithm

Suppose we have a congruence class set $\mathcal{C}$ that is closed and in normal form. The precise version of the matching problem that we would like to solve is: given classes C_i and C_j, what variable instantiations σ allow C_i to match to C_j? A particularly important example of this problem for deductive proving occurs when C_i contains the goal term and C_j is the true class, when the question is equivalent to asking what instantiations of variables in the goal term make it true. First we define these terms, and then present a solution.

3.1 Definitions

We define a **variable instantiation** $\sigma : V \rightarrow \mathcal{T}$ to be a function from logic variables to terms (which may themselves contain logic variables). Given a term $t \in \mathcal{T}$, we write σt to mean the instantiation of variables in t according to σ.

For a congruence class set $\mathcal{C}$ containing classes C_i and C_j, we say that σ **allows** $\mathbf{C}_i$ **to match to** $\mathbf{C}_j$ if

$$\exists\, t_i\, t_j .\ t_i \in_{\mathcal{C}} C_i \ \wedge\ t_j \in_{\mathcal{C}} C_j \ \wedge\ \sigma t_i = t_j$$

We define a function $\chi : V \times \mathcal{C} \rightarrow \mathcal{P}(V \rightarrow \mathcal{T})$ from variable-class pairs to sets of variable instantiations

$$\chi(X, C) = \{\sigma : V \rightarrow \mathcal{T} \ \mid\ \sigma(X) \in_{\mathcal{C}} C\}$$

and let ϕ be

$$\phi(S) = \bigcap_{(X,C) \in S} \chi(X, C)$$

So $\phi(S)$ is the set of all functions from V to $\mathcal{T}$ that, for every (X, C) pair in S, map variable X to a term 'provably in C'. We call sets of variable-class pairs **substitutions**.

3.2 The Algorithm

The reason that we defined the function ϕ in the previous subsection is that for every pair (C_i, C_j) of classes the algorithm we present finds substitutions S such that every variable instantiation in $\phi(S)$ allows C_i to match to C_j. Let $match(C_i, C_j)$ be the current set of substitutions between (C_i, C_j).

We initialize each $match(C_i, C_j)$ as follows:

1. First set $match(C_i, C_j) = \{\}$.
2. If rep_i and rep_j have different HOL types, then finish.
3. For every logic variable X with $class(X) = C_i$, add $\{(X, C_j)\}$.
4. If $i = j$ then add $\{(X, class(X)) \mid X \text{ a logic variable in } rep_i\}$.

And now we inductively build up *match*, terminating when it is no longer possible to add anything new to any $match(C_i, C_j)$:

- Given an element in class C_j of the form $lambda(v, rep_b)$ and a substitution $S \in match(C_B, C_b)$; if there exists an element $lambda(v, rep_B)$ in some class C_i then add S to $match(C_i, C_j)$.
- Given an element in class C_j of the form $app(rep_f, rep_a)$, and substitutions $S \in match(C_F, C_f)$ and $S' \in match(C_A, C_a)$; if both
  - S and S' are **compatible**, (i.e., for every variable X there is at most one element in $S \cup S'$ of the form (X, C)).
  - there exists an element $app(rep_F, rep_A)$ in some class C_i

  then add $S \cup S'$ to $match(C_i, C_j)$.

Note that here we are using the fact that $\mathcal{C}$ is in normal form.

We will illustrate the algorithm on an example. Unfortunately, all real-life examples are too large to represent, so the example will necessarily have to be rather artificial. We invent an 'if-and-only-if' operator, f, which returns true exactly when its two boolean arguments are equal. The example shows what matches occur on the term $f \top (f \top \top)$.[6]

1. Here is the result of adding the fact $f\ X\ X$ and the term $f \top (f \top \top)$ to a minimal class set:

$$\begin{array}{llllll}
\{ & \top & \{ & \top, & f\ X\ X & \}, \\
 & \bot & \{ & \bot & & \}, \\
 & f & \{ & f & & \}, \\
 & X & \{ & X & & \}, \\
 & f\ X & \{ & f\ X & & \}, \\
 & f\ \top & \{ & f\ \top & & \}, \\
 & f\ \top\ \top & \{ & f\ \top\ \top & & \}, \\
 & f\ \top\ (f\ \top\ \top) & \{ & f\ \top\ (f\ \top\ \top) & & \} \quad \}
\end{array}$$

2. The initial matches are easy to calculate, every class matches itself, and the boolean logic variable X matches all the boolean classes:

[6] In this section we use curried notation for function application, because our underlying method uses only *lambda* and *app* to construct terms.

$$
\begin{array}{rcl}
match(\top,\ \top) & = & \{\{\}\} \\
match(\bot,\ \bot) & = & \{\{\}\} \\
match(f,\ f) & = & \{\{\}\} \\
match(X,\ X) & = & \{\{(X,\ X)\}\} \\
match(f\ X,\ f\ X) & = & \{\{(X,\ X)\}\} \\
match(f\ \top,\ f\ \top) & = & \{\{\}\} \\
match(f\ \top\ \top,\ f\ \top\ \top) & = & \{\{\}\} \\
match(f\ \top\ (f\ \top\ \top),\ f\ \top\ (f\ \top\ \top)) & = & \{\{\}\} \\
match(X,\ \top) & = & \{\{(X,\ \top)\}\} \\
match(X,\ \bot) & = & \{\{(X,\ \bot)\}\} \\
match(X,\ f\ \top\ \top) & = & \{\{(X,\ f\ \top\ \top)\}\} \\
match(X,\ f\ \top\ (f\ \top\ \top)) & = & \{\{(X,\ f\ \top\ (f\ \top\ \top))\}\}
\end{array}
$$

All other *match* sets are empty.

3. After one inductive step, we gain the following extra match:

   $$match(f\ X,\ f\ \top) \quad = \quad \{\{(X,\ \top)\}\}$$

   This comes from the application inductive step, using the compatible substitutions $\{\}$ and $\{(X, \top)\}$, contained in $match(f, f)$ and $match(X, \top)$ respectively.

4. After two inductive steps, we gain the following extra match:

   $$match(\top,\ f\ \top\ \top) \quad = \quad \{\{(X,\ \top)\}\}$$

   Again from the application inductive step: this uses the substitution from the previous step, and the compatible substitution $\{(X, \top)\}$ contained in $match(X, \top)$. Since $f\ X\ X$ has representative $\top$, this is how the match appears. Notice that there are no matches from $\top$ to $f\ \top\ (f\ \top\ \top)$, because the relevant substitutions are incompatible.

5. The algorithm now terminates, because no more substitutions can be added to any *match* set. All the match sets are returned to the calling application.

We will return to this example in Section 4, to show how the percolation algorithm makes use of the returned match sets.

There are two significant optimizations that can be made to the theoretical algorithm. Firstly, we do not add a substitution S that is less general than a match S' that we already have (i.e., $\phi(S) \subseteq \phi(S')$), though the algorithm above adds anything that is different. This costs nothing and promotes faster convergence.

Secondly, we divide the inductive stage into passes, so that on pass $n + 1$ we add matches that arise from the result of pass n. This allows us to optimize by considering only the matches that were new at pass n, instead of every match. This is most significant for the application inductive step, where if there are m new and M old matches at pass n, it only has to consider $2mM + m^2$ combinations instead of $(m + M)^2$.

3.3 Proofs

We will prove three theorems about the algorithm: firstly, it terminates on all inputs; secondly, a soundness result that every match that it finds is valid; thirdly, a completeness result that every possible match is found. The purpose of these theorems is

to show that the theoretical algorithm is logically sound[7] and also to provide some useful information on the scope of the ideas. They can be used in proving facts about procedures that make use of the Matching Algorithm: we will see in Section 4 that the Percolation Algorithm relies on the termination and completeness results.

Theorem 3.1 (Termination) The matching algorithm terminates on all inputs.

PROOF. Let c be the number of classes in $\mathcal{C}$, and let v be the number of logic variables in $\mathcal{C}$. Note that the algorithm does not create any new classes or logic variables, so c and v are fixed.

The number of variable-class pairs that may arise in the matching algorithm is bounded above by vc, and so the number of possible sets of variable-class pairs, i.e., substitutions, is bounded above by 2^{vc}.

Define the termination measure to be the sum of the number of substitutions in all the *match* sets. Since there are c^2 *match* sets, this is bounded above by $c^2 2^{vc}$.

The algorithm terminates when it cannot add any new substitutions to any *match* set, so the termination measure must increase on every inductive step. But it is bounded above, so the algorithm must terminate. ■

Lemma 3.2 For all substitutions S and S', we have that $\phi(S \cup S') = \phi(S) \cap \phi(S')$.

PROOF.

$$\begin{aligned} \phi(S \cup S') &= \bigcap_{(X,C) \in S \cup S'} \chi(X, C) \\ &= \left(\bigcap_{(X,C) \in S} \chi(X, C) \right) \cap \left(\bigcap_{(X,C) \in S'} \chi(X, C) \right) \\ &= \phi(S) \cap \phi(S') \end{aligned}$$

■

Theorem 3.3 (Soundness) For every substitution $S \in match(C_i, C_j)$, for every variable instantiation $\sigma \in \phi(S)$, we have that σ allows C_i to match to C_j.

PROOF. We proceed by structural induction on *match*, and assume that we have just added substitution S to $match(C_i, C_j)$, and that $\sigma \in \phi(S)$.

Firstly suppose S is one of the initial substitutions; for step 3 it is clear from the definitions that $X \in_{\mathcal{C}} C_i$ and $\sigma(X) \in_{\mathcal{C}} C_j$, similarly for step 4 we have that $rep_i \in_{\mathcal{C}} C_i$ and $\sigma rep_i \in_{\mathcal{C}} C_i$.

Now consider the lambda inductive step. By the induction hypothesis there must exist $t_B \in_{\mathcal{C}} C_B$ and $t_b \in_{\mathcal{C}} C_b$ with $\sigma t_B = t_b$, so therefore we have that $lambda(v, t_B) \in_{\mathcal{C}} C_i$, $lambda(v, t_b) \in_{\mathcal{C}} C_j$ and $\sigma lambda(v, t_B) = lambda(v, t_b)$.

Finally consider the application inductive step. There must exist classes C_F, C_f, C_A and C_a such that $app(rep_F, rep_A)$ is an element in C_i and $app(rep_f, rep_a)$ is an element in C_j. In addition, there must exist compatible substitutions $S_f \in match(C_F, C_f)$ and $S_a \in match(C_A, C_a)$ such that $S = S_f \cup S_a$, and so by Lemma 3.2

[7] Though in `hol98`, the design of the theorem-prover forces all inferences to be expressed as combinations of primitive inferences and these are checked by the logical kernel, so in this environment we are protected against proving false theorems.

we know $\sigma \in \phi(S_f)$ and $\sigma \in \phi(S_a)$. Now we may apply the induction hypothesis to get $t_F \in_{\mathcal{C}} C_F$, $t_f \in_{\mathcal{C}} C_f$, $t_A \in_{\mathcal{C}} C_A$, $t_a \in_{\mathcal{C}} C_a$ with $\sigma t_F = t_f$ and $\sigma t_A = t_a$. Therefore we have that $app(t_F, t_A) \in_{\mathcal{C}} C_i$, $app(t_f, t_a) \in_{\mathcal{C}} C_j$, and $\sigma\ app(t_F, t_A) = app(t_f, t_a)$, as required. ∎

Lemma 3.4 For every class C_i, all the logic variables in rep_i are also in every element of C_i.

Proof. This is true when the class initializes, and the case when two classes merge is taken care of in the footnote to Subsection 2.3. ∎

Lemma 3.5 For every $t \in_{\mathcal{C}} C$, there is an element x in C such that one of the following holds:

- t is a logic variable, constant or normal variable and $t = x$.
- $t = lambda(v, t_b)$, $x = lambda(v, rep_b)$ and $t_b \in_{\mathcal{C}} C_b$.
- $t = app(t_f, t_a)$, $x = app(rep_f, rep_a)$, $t_f \in_{\mathcal{C}} C_f$ and $t_a \in_{\mathcal{C}} C_a$.

Proof. Consider a counter-example t with minimal term depth. $t =_{\mathcal{C}} rep$, so consider the first time we meet an element x in C in the rewriting chain between t and rep. If x is a logic variable, constant or normal variable then we must have that $t = x$ because there can be no previous step in the chain. If x is a lambda or application term, then it must always have been so, and now we can appeal to the minimality of t to show that the proper subterms must satisfy the necessary conditions. ∎

Theorem 3.6 (Completeness) For every variable instantiation σ that allows C_i to match to C_j, there exists a substitution $S \in match(C_i, C_j)$ such that $\sigma \in \phi(S)$.

Proof. Assume the result is false and pick a counterexample σ. There must exist $t_i \in_{\mathcal{C}} C_i$ and $t_j \in_{\mathcal{C}} C_j$ with $\sigma t_i = t_j$, and we may assume that the term depth of t_i is minimal over all counterexamples. Using Lemma 3.5 we perform a case split on t_i:

Suppose t_i is a logic variable X. Then we will have that $\sigma \in \phi(\{(X, C_j)\})$, and since the HOL type of t_i is the same as the HOL type of σt_i this substitution was added to the set in step 3 of the initialization. Contradiction.

Suppose t_i is a constant or normal variable. Then $t_i = t_j$, and since $\mathcal{C}$ is closed we have that $C_i = C_j$. By Lemma 3.4 the representative of C_i contains no logic variables at all, and so the set $\{\}$ was added to $match(C_i, C_i)$ in step 4 of the initialization. $\phi(\{\})$ contains every variable instantiation, so certainly contains σ. Contradiction.

Suppose $t_i = lambda(v, t_B)$, so $t_j = lambda(v, t_b)$, and we must have classes such that $t_B \in_{\mathcal{C}} C_B$ and $t_b \in_{\mathcal{C}} C_b$. Since $\sigma t_B = t_b$ and t_B has strictly smaller term depth than t_i (by the normal form property), we must have a substitution $S \in match(C_B, C_b)$ such that $\sigma \in \phi(S)$. Therefore S must have been added to $match(C_i, C_j)$ in the lambda inductive step. Contradiction.

Finally suppose $t_i = app(t_F, t_A)$, so $t_j = app(t_f, t_a)$, and we must have classes such that $t_F \in_{\mathcal{C}} C_F$, $t_A \in_{\mathcal{C}} C_A$, $t_f \in_{\mathcal{C}} C_f$ and $t_a \in_{\mathcal{C}} C_a$. Since $\sigma t_F = t_f$ and t_F has strictly smaller term depth than t_i, we must have a substitution $S \in match(C_F, C_f)$ such that $\sigma \in \phi(S)$. Similarly we must have $S' \in match(C_A, C_a)$ such that $\sigma \in \phi(S')$. By Lemma 3.2 we have that $\phi(S) \cap \phi(S') = \phi(S \cup S')$, so $\sigma \in \phi(S \cup S')$, and hence S and S' are compatible. This implies that $S \cup S'$ was added to $match(C_i, C_j)$ in the application inductive step. Contradiction. ∎

4 Percolation Algorithm

The Percolation Algorithm performs undirected rewriting on the congruence classes, and is an easy application of the Matching Algorithm. Here is how it works:

1. Perform the Matching Algorithm.
2. For every substitution S in $match(C_i, C_j)$ and for every element t in C_i, create the element t' by applying the substitution S to t, and add t' to C_j. This step is illustrated in Figure 1.
3. Perform a close operation.

This is one iteration of the percolation algorithm. There is no hope here of always reaching a fixed-point after many iterations; this would allow us to decide the word problem for the equalities in $\mathcal{C}$, which is undecidable [3]. In general our algorithm is a semi-decision procedure for the word problem, and in particular cases where a fixed-point is reached it is a full decision procedure.

Returning to the example used in Section 3 to illustrate the matching algorithm, suppose the percolation algorithm had performed step 1 and been returned the *match* sets we created in the example. In step 2 it would be able to make exactly one addition: adding $\top$ to the class with representative $f \top \top$. After the close operation in step 3, this is how the classes look:

$$
\begin{array}{lllllll}
\{ & \top & \{ & \top, & f\ X\ X, & f\ \top\ \top & \}, \\
 & \bot & \{ & \bot & & & \}, \\
 & f & \{ & f & & & \}, \\
 & X & \{ & X & & & \}, \\
 & f\ X & \{ & f\ X & & & \}, \\
 & f\ \top & \{ & f\ \top & & & \} \quad \}
\end{array}
$$

In practice we give each element a level number, and we define the level of a substitution S in $match(C_i, C_j)$ to be the maximum level of: the element in C_j that was matched; and the substitutions used to create S. When we add new elements to C_j we give them the level of the match plus one. Now if we insist that the level of all elements added is less than some maximum then we can run the percolation algorithm until a fixed-point is reached, since now it is guaranteed to exist. This

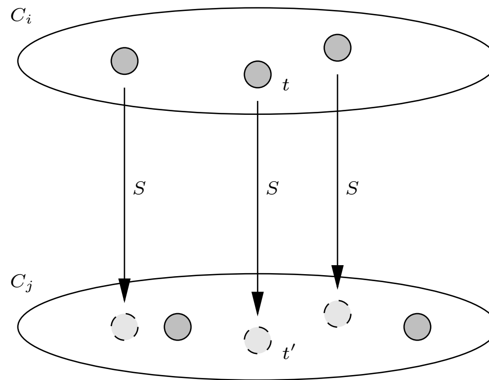

FIG. 1. One Step of the Percolation Algorithm.

method of allocating level numbers was chosen so that we can give our rewrite rules (like $X+Y=Y+X$) an initially high level number, which will then stop them being rewritten like everything else!

5 Results

The examples we chose to test the program are listed in Table 1, and the results for these examples are detailed in Table 2. The Percolation Algorithm is not the most efficient way to prove the test theorems, and gets bogged down when the maximum level is set above about 4 (depending on which rewriting theorems are used). The first example is a triviality that is impossible for standard congruence closure [3]; the second example (from the introduction) derives the right inverse law from the group axioms; the other examples depend on the usual arithmetic rewrites and are designed to show what happens in such a rich theory. We give each example in Table 1 a number which is used in Table 2, and also show the minimum level necessary to prove the result. The columns in Table 2 in order are: example number, iteration number, number of classes, number of elements, number of matches, time to find the matches, time to add the new elements (assimilation), time to perform congruence closure, and total time spent on the iteration. We measured the number of elements and classes before each iteration. At the end of all the iterations we write in bold the total time spent in each phase.[8] Note that the standard first order prover in HOL, `MESON_TAC`, proves examples 1, 3 and 4 in under 2 seconds, but can't prove the others at all (at least, not in any reasonable time).

The first thing that can be noticed from Table 2 is that the times to perform the three phases are of the same order: no one phase relatively dominates or vanishes. The second thing is that there is an awfully large number of matches for even small maximum levels, bringing in a correspondingly large number of new classes and elements. The congruence closure phase can reduce the size of the class set by a factor of 10 after an assimilation, and if we weren't using congruence classes this reduction in space would not be possible. However, despite this saving, undirected rewriting causes a subterm explosion; all congruence classes can really do is postpone the point beyond which we lose control.

6 Conclusions

We have described the technical steps involved in integrating logic variables with congruence classes, and presented an algorithm to discover matches between classes. This problem is analogous to the problem of finding the shortest path between two points on a weighted graph: in both cases the solution algorithm computes seemingly more than is required. Dijkstra's Algorithm [7] solves the shortest path problem but must compute the shortest path from the source point to every other point; and the matching algorithm presented here computes all matches between every pair of classes.

This 'side-effect' is exploited in the Percolation Algorithm: finding every match

---

[8] All times are in seconds, using version Taupo 2 of `hol98` running on an Intel Pentium III 600MHz. The memory use for the examples is less than 10Mb, and garbage collection accounts for about 10% of the matching time, 5% of the assimilation time, and 20% of the closing time.

is exactly what we want for undirected rewriting, and congruence classes provide an efficient storage mechanism. However, the results show that it is not efficient enough to stem the tide in theories such as arithmetic with large sets of equalities, and it cannot compete with a rewriter when there exists a directed rewrite set. This suggests a natural compromise: for all equalities that make up such a rewrite set, let the rewriter work and feed the resulting simplifications to the congruence classes, we can then run the Percolation Algorithm on these and the rest of the equalities.

This further integration will form the basis of future work, as well as the original aim of building a deductive proof procedure on top of the congruence classes. Unification is frequently used in deductive proof search, and perhaps there exists a unification algorithm on classes analogous to the matching algorithm presented in this paper. We have briefly investigated this, and the problem seems to be more difficult: at the separation of variables phase we may have to create new classes to accommodate the new variables, so even our termination proof fails in the new context. There are also some improvements to be made to the current congruence class component, including allowing type variables to match (which will be useful in polymorphic theories), and allowing higher-order matching by including conversions on the underlying lambda calculus. This could perhaps work in a similar way to the Percolation Algorithm, looking for terms of the form $app(lambda(v, C_i), C_j)$, and then producing a new term by replacing all occurrences of v in C_i with C_j.

7 Related Work

McAllester [11] has also used congruence classes for storing terms, in order to perform rewriting in non-confluent theories. The application is different but the theory is very similar, since it is motivated by the same goal of reducing the space needed to store terms. One notable difference is that McAllester uses context-free grammars to represent the current state of the congruence classes, whereas we use an ad-hoc representation. However, in higher-order logic this doesn't really matter, because there are only two term constructors. Another difference is that in our system the equations for rewriting come from the congruence classes, whereas in McAllester's setup they are prescribed in advance. This is in keeping with our respective applications, McAllester is seeking to improve rewriting performance, and we want to improve the handling of equality in deductive proof procedures.

Apart from this, the most frequent use of congruence classes has been in the heart of the Nelson-Oppen combination of decision procedures [12, 13], which uses congruence closure to propagate the equalities generated by the component decision procedures. Since traditional congruence closure treats term variables as constants, most of the work in equality reasoning has concentrated on rewriting. There is much theory on this (see Baader & Nipkow [3] for a comprehensive overview), and many systems (powerful simplifiers exist in many theorem provers, including HOL[9], Isabelle[10] and PVS[11]; and the OBJ family of languages is based on rewriting logic [8]).

There have been many attempts to integrate equality reasoning with deductive proving, with varying degrees of sophistication. The first-order prover Gandalf [14]

[9] `http://www.ftp.cl.cam.ac.uk/ftp/hvg/hol98/`

[10] `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`

[11] `http://www.csl.sri.com/sri-csl-pvs.html`

includes paramodulation as a basic proof step, as do many others, while Harrison's implementation in HOL of Model Elimination [9] axiomatizes equality and relies on deductive proof alone. A comparison of the two [10] suggests that paramodulation is more effective than pure deduction. In this paper we have argued that even more infrastructure would improve performance again.

On the more general note of interleaving equality and deductive steps, we observe that Boyer and Moore [6] have argued for (and implemented) a tight integration of decision procedures and provers; Armando and Ranise's reconstruction [2] separated the sharing of data but preserved the mutual recursion; and Zammit [15] made use of rewriting interleaved with deductive proof steps by generalizing proof rules with simplifiers.

Acknowledgements

I had many fruitful discussions with Konrad Slind, and my supervisor, Mike Gordon, while engaging on this research. Donald Syme and Michael Norrish also gave me a helping hand to produce the work. The comments of the anonymous referees of the TPHOLs conference and the Journal of the IGPL improved the paper enormously.

References

[1] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] A. Armando and S. Ranise. From integrated reasoning specialists to "plug-and-play" reasoning components. In *Proc. of the 2nd Intl. Workshop on First Order Theorem Proving (FTP '98)*, volume 1476 of *Lecture Notes in Computer Science*, pages 42–54. Springer, 1998.

[3] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] Richard J. Boulton. Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, 3(1/2):25–47, August 1993.

[5] Richard J. Boulton. Combining decision procedures in the HOL system. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 75–89, Aspen Grove, UT, USA, September 1995. Springer-Verlag.

[6] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[7] Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

[8] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, Mass., 1st edition, 1996.

[9] John Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.

[10] Joe Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999.

[11] David McAllester. Grammar rewriting. In Deepak Kapur, editor, *Automated Deduction, CADE-11: 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15–18, 1992: Proceedings*, number 607 in Lecture Notes in Artificial Intelligence, pages 124–138. Springer-Verlag, 1992.

[12] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[13] Greg Nelson and Derek C. Oppen. Fast decision procedures bases on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[14] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, April 1997.

[15] Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent at Canterbury, March 1999.

A HOL Implementation

A.1 Notation

Terms in HOL are either constants, variables, function applications of terms to terms ($app(t_1, t_2)$), or lambda abstractions ($lambda(v, t)$). Thus `5 + 3` is really app(`$+ 5`, `3`), which in turn is app(app(`$+`, 5), 3). Note that infix operators gain the prefix `$` when they are not in their usual infix position.

The HOL constants `T` and `F` represent the logical *true* and *false*. We use HOL variables called `lv0`, `lv1`, `lv2`, ... to denote logic variables.

HOL theorems can be created only using primitive inferences, and are printed in the form `[..] |- f lv3 = lv3`. The `|-` represents the logical $\vdash$ to mean syntactic derivability of the conclusion on the right from the assumptions on the left. `[..]` represents two assumptions, which by default HOL does not show explicitly.

There is a fixed set of HOL primitive inferences, and some that are relevant to this paper are: `ASSUME`, `REFL`, `SYM`, `TRANS`, and `MK_COMB`.

- `ASSUME` takes a term t and returns the theorem $t \vdash t$.
- `REFL` takes a term t and returns the theorem $\vdash t = t$.
- `SYM` takes a theorem of the form $\Gamma \vdash t_1 = t_2$ and returns the theorem $\Gamma \vdash t_2 = t_1$.
- `TRANS` takes two theorems of the form $\Gamma \vdash t_1 = t_2$ and $\Delta \vdash t_2 = t_3$, and returns the theorem $\Gamma \cup \Delta \vdash t_1 = t_3$.
- `MK_COMB` takes two theorems of the form $\Gamma \vdash f_1 = f_2$ and $\Delta \vdash a_1 = a_2$, and returns the theorem $\Gamma \cup \Delta \vdash app(f_1, a_1) = app(f_2, a_2)$.

A.2 Congruence Classes

To implement congruence classes in HOL we define the following ML datatypes:

```
datatype elt = Elt of term * thm;
datatype class = Class of term * elt list;
```

Note firstly that `type1 * type2` is the way ML represents a cartesian product type. The `term` in the `class` type is the class representative; within `elt` the `term` is the class element and the `thm` is the equality theorem ‘the element is equal to the class representative’. The reason we include the term separately when it is always the left hand side of the theorem is to speed up access.

To give a feeling for the set-up, we give a simple example. Here is the initial class set to which we will be adding terms:

```
- classes_null;
> val it =
    [Class(T, [Elt(T,  [] |- T = T)]),
     Class(F, [Elt(F,  [] |- F = F)])]
    : class list
```

Now we are going to add a fact to the class set using `introduce_fact`, a function which adds the conclusion of the theorem to the class set (involving a conversion to normal form), then performs a close with the information that the fact should be in the true class.

```
- val cs = introduce_fact (ASSUME ‘f (f (f lv3)) = (lv3:'a)‘) classes_null;
> val cs =
    [Class(lv3, [Elt(lv3, |- lv3 = lv3),
                 Elt(f (f (f lv3)),  [.] |- f (f (f lv3)) = lv3)]),
```

```
    Class(f lv3, [Elt(f lv3, |- f lv3 = f lv3)]),
    Class(f (f lv3), [Elt(f (f lv3), |- f (f lv3) = f (f lv3))]),
    Class(f, [Elt(f, |- f = f)]),
    Class($=, [Elt($=, |- $= = $=)]),
    Class(T, [Elt(lv3 = lv3,  [.] |- (lv3 = lv3) = T),
              Elt(T, |- T = T)]),
    Class($= lv3, [Elt($= lv3,  [.] |- $= lv3 = $= lv3)]),
    Class(F, [Elt(F, |- F = F)])]
   : class list
```

Note that we have precisely one logic variable (`lv3`); the others are normal variables. We now add another fact which produces the following collapse:

```
- val cs' = introduce_fact (ASSUME `f (f (f (f (f lv3)))) = (lv3:'a)`) cs;
> val cs' =
   [Class(lv3, [Elt(lv3, |- lv3 = lv3),
                Elt(f lv3,  [..] |- f lv3 = lv3)]),
    Class(T, [Elt(lv3 = lv3,  [.] |- (lv3 = lv3) = T),
              Elt(T, |- T = T)]),
    Class($=, [Elt($=, |- $= = $=)]),
    Class(f, [Elt(f, |- f = f)]),
    Class($= lv3, [Elt($= lv3,  [.] |- $= lv3 = $= lv3)]),
    Class(F, [Elt(F, |- F = F)])]
   : class list
```

Finally we add a new term to the class set:

```
add_term `f (f (f (f (lv3:'a))))` cs';
> val it =
    ( [..] |- f (f (f (f lv3))) = lv3,
     [Class(lv3,
       ...same classes as last time...
      Class(F, [Elt(F, |- F = F)])])
   : Thm.thm * class list
```

Notice the theorem that we get back in addition to the new class set, which tells us the normal form.

A.3 Matching Algorithm

In our implementation, we combine all the $match(C_i, C_j)$ sets for each class C_j, and define a `match` datatype to store the substitutions:

```
datatype match = Match of (term * (int * term) list) * (thm * thm);
```

The first `term` is rep_i, and the `(int * term) list` represents a substitution (logic variables are numbered, and the `term` is the class representative). Finally the first theorem is of the form $t = rep_i$ and the second is $t' = rep_j$, where if we apply the substitution to t we will get t'. We need this to prove to HOL that the substitution is really valid.

Initializing the *match* sets is easily implemented, and we will briefly comment on the method for the application inductive step, as illustrated in Figure 2. We assume there are compatible substitutions S taking F to f and S' taking A to a. We have the term $app([F],[A])$[12] in class C_i and term $app([f],[a])$ in class C_j, and want to construct the application match. The two theorems we need are of the form $t = [app([F],[A])]$ and $t' = [app([f],[a])]$, for any t and t' such that the substitution $S \cup S'$ takes t to t'. If we choose $t = app(F, A)$ and $t' = app(f, a)$, then the required two theorems can be created like this:

$$\begin{aligned} E_t &= \texttt{TRANS}\ (\texttt{MK_COMB}\ (E_F, E_A))\ E_i \\ E'_t &= \texttt{TRANS}\ (\texttt{MK_COMB}\ (E_f, E_a))\ E_j \end{aligned}$$

Note that we are using HOL theorems at every stage, which may result in a bottleneck in the logical kernel. Boulton [4] provides a lazy-theorem approach to avoid such problems if they arise.

Received 8 June 2000. Revised 9 August 2000

[12] Recall that $[X]$ is the notation for the class representative of the term X.

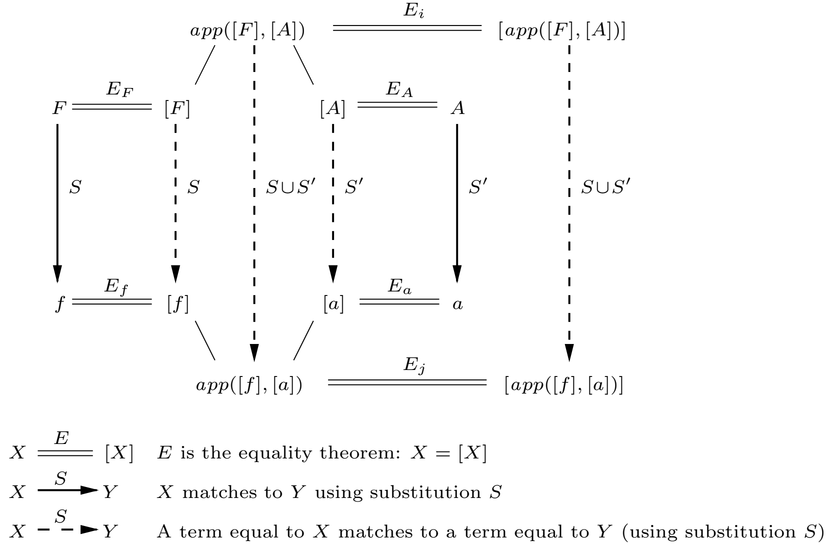

FIG. 2. Matching through an application.

Example	**Level**	**Theorem**
1	1	$(\forall x.\ f(f(x)) = g(x)) \Rightarrow (f(g(a)) = g(f(a)))$
2	2	$(\forall x\, y\, z.\ ((x * y) * z = x * (y * z)) \wedge (e * x = x)$ $\wedge\, (i(x) * x = e)) \Rightarrow (x * i(x) = e)$
3	1	$abc = cba$
4	2	$abcd = dcba$
5	2	$abcde = edcba$
6	3	$(a+1)(a+1) = aa + a + a + 1$

TABLE 1. Examples Chosen to Test the Percolation Algorithm.

Ex	It	#C	#E	#M	Match	Assim.	Close	Total
1	1	17	21	26	0.003	0.005	0.007	0.015
	2	15	22	24	0.004	0.005	-	0.009
					0.007	**0.010**	**0.007**	**0.024**
2	1	28	36	46	0.008	0.011	0.012	0.031
	2	28	40	126	0.040	0.125	0.069	0.234
	3	67	101	754	0.738	1.034	0.121	1.893
	4	85	134	857	1.308	1.283	0.079	2.670
	5	76	122	994	1.962	1.138	0.088	3.188
	6	77	129	951	1.738	1.242	0.068	3.048
	7	80	145	857	1.578	1.179	-	2.757
					7.372	**6.012**	**0.437**	**13.821**
3	1	51	70	109	0.027	0.054	0.075	0.156
	2	59	98	118	0.035	0.054	-	0.089
					0.062	**0.108**	**0.075**	**0.245**
4	1	56	75	143	0.033	0.083	0.108	0.224
	2	76	129	1047	0.499	1.348	2.633	4.480
	3	189	401	1168	0.826	1.169	-	1.995
					1.358	**2.600**	**2.741**	**6.699**
5	1	61	80	177	0.045	0.126	0.161	0.332
	2	89	154	1413	0.665	2.053	6.289	9.007
	3	188	477	1562	1.074	1.976	-	3.050
					1.784	**4.155**	**6.450**	**12.389**
6	1	53	72	131	0.033	0.092	0.076	0.201
	2	69	114	770	0.399	1.034	0.957	2.390
	3	83	179	3507	5.929	8.950	10.914	25.793
	4	96	228	2609	5.617	4.995	0.607	11.219
	5	200	406	2501	5.403	6.837	-	12.240
					17.381	**21.908**	**12.554**	**51.843**

TABLE 2. Detailed Profiles of the Percolation Algorithm on the Examples.