Proof-producing Congruence Closure

Robert Nieuwenhuis* and Albert Oliveras**

Technical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain
`www.lsi.upc.es/~roberto` `www.lsi.upc.es/~oliveras`

Abstract. Many applications of congruence closure nowadays require the ability of recovering, among the thousands of input equations, the small subset that caused the equivalence of a given pair of terms. For this purpose, here we introduce an incremental congruence closure algorithm that has an additional *Explain* operation.
First, two variations of union-find data structures with *Explain* are introduced. Then, these are applied inside a congruence closure algorithm with *Explain*, where a k-step proof can be recovered in almost optimal time (quasi-linear in k), without increasing the overall $O(n \log n)$ runtime of the fastest known congruence closure algorithms.
This non-trivial (ground) equational reasoning result has been quite intensively sought after (see, e.g., [SD99,dMRS04,KS04]), and moreover has important applications to verification.

1 Introduction

Union-find data structures maintain the *equivalence* relation induced by a given sequence of *Union* operations between pairs of elements. Similarly, *congruence closure* algorithms maintain a *congruence* relation given by a sequence of pairs of *terms* (i.e., equations) without variables. The difference between equivalence closure and congruence closure is that the congruence relation, in addition to reflexivity, symmetry and transitivity, also satisfies the *monotonicity* axioms saying, for all f, that $f(x_1 \ldots x_n){=}f(y_1 \ldots y_n)$ whenever $x_i{=}y_i$ for all i in $1 \ldots n$.

Example 1. The equation $a{=}b$ belongs to the congruence generated by the three equations: $b{=}d$, $f(b){=}d$, and $f(d){=}a$.

This is equivalent to saying that $a{=}b$ is a logical consequence (in first-order logic with equality) of these three ground equations. □

Congruence closure is closely related to ground Knuth-Bendix completion; in fact, as usual, our congruence closure algorithm applied to an input set of ground equations E builds a convergent term rewrite system for E (possibly with some new symbols).

* Both authors partially supported by Spanish Min. of Educ. and Science by the LogicTools project (TIN2004-03382)
** Supported by FPU grant AP2002-3533 from the Spanish MECD Ministry.

Decision procedures based on congruence closure are used in numerous deduction and verification systems, where the generation of *explanations* is highly desirable if not required. For instance, this is crucial in the so-called *lazy* approaches to decision procedures for Boolean formulae over theory atoms. In these procedures, the Boolean formulae frequently include equality atoms; see, e.g., CVC-Lite, at `verify.stanford.edu/CVCL,` and [dMR02,ABC$^+$02,BDS02,FJOS03]. All these approaches are *lazy* in the sense that initially each equality atom is simply abstracted by considering it as a distinct propositional variable, and the resulting propositional formula is sent to a SAT solver. If the SAT solver reaches a (partial) model that is not a congruence, an additional propositional clause (a *lemma*) precluding that model is added; this is iterated (*many* times) until the SAT solver finds a congruence model or all assignments have been explored.

Example 2. Assume that, in such a lazy approach, the model being built by the SAT solver is fed into the congruence closure algorithm as a (long!) sequence of atoms that, in particular, includes $b = d$, $f(b) = d$, and $f(d) = a$. Then, if additionally $a \neq b$ is given, it is no longer a congruence (see Example 1).

At that point, the congruence closure algorithm has to generate as a lemma the clause $b{=}d \wedge f(b){=}d \wedge f(d){=}a \longrightarrow a{=}b$, because the first three atoms are the explanation of $a{=}b$. It is hence crucial in these applications to efficiently recover this small explanation among the (thousands of) originally input equations. □

Another recent approach for the flexible generation of decision procedures given in [GHN$^+$04] also heavily relies on incremental congruence closure with intermixed *Explain* operations. The basic idea is similar to the *CLP*(X) scheme for constraint logic programming: to provide a clean and efficient integration of specialized theory solvers within the Davis-Putnam-Logemann-Loveland procedure [DP60,DLL62]. A general engine DPLL(X) is used, where X can be instantiated with a solver for a given theory T, thus producing a system DPLL(T). Each time the DPLL(T) procedure produces a conflict, explanations need to be generated by the theory solver for building the *conflict graph* that is used for *non-chronological backtracking* in modern SAT solvers like Chaff [MMZ$^+$01]. The fact that this approach currently outperforms previous techniques on logics with equality is largely due to the efficient incremental algorithm for congruence closure with explanations described here (see [GHN$^+$04] for details about the DPLL(T) approach and experiments on benchmarks from a large variety of verification problems).

Since in such an incremental setting many *Explain* operations occur during a single congruence closure procedure, it is crucial to efficiently recover these explanations, even at the expense of making the congruence closure algorithm slightly slower in practice. If the congruence closure procedure deals with input equations of size n and *Explain*, say, were linear in n, the cost of *Explain* would enormously dominate the $O(n \log n)$ runtime of the overall congruence closure algorithm. Here we present, to our knowledge, the first congruence closure algorithm able to produce these explanations in an efficient way.

Section 2 of this paper is on union-find data structures with *Explain*. Indeed, already for union-find data structures the problem requires some thinking, since the information about the original input unions is, in general, lost in the compact representations of the equivalence relation. We first very briefly introduce some basic notions about union-find data structures and define the *Explain* operation. A first solution that supports optimal *Union* and *Find* operations (as in Tarjan's well-known algorithm with path compression [Tar75]) and recovers the k-step proof in time $O(k \log n)$ is given in Subsection 2.1. In Subsection 2.2 we describe another union-find data structure that has optimal $O(k)$ *Explain* operations and optimal *Find*, at the expense of a slightly more costly *Union*, which has an amortized time bound of $O(\log n)$.

Section 3 is the core of this paper, where the latter union-find data structure is applied inside an incremental congruence closure algorithm. Its complexity is analyzed in Subsection 3.3, where we show that the use of this more costly union-find algorithm (needed for bookkeeping the explanations) does not increase the overall $O(n \log n)$ runtime of the fastest known congruence closure algorithms.

The *Explain* operation is given in Subsection 3.4, and analyzed in detail in Subsection 3.5, showing that it is almost optimal, running in $O(k\, \alpha(k, k))$ time for a k-step explanation, where $\alpha(k, k)$ (related to the inverse of Ackermann's function) is in practice never larger than 4. Subsection 3.6 discusses quality issues of explanations, gives extensive experimental results, and introduces several extensions with practical impact of our explanation algorithms.

2 Union-find with Proofs

For the sake of self-containedness of this paper and for introducing some notation, here we first shortly explain the classical union-find data structure (see, e.g., [CLR90] for details). A binary *relation* over a set $\mathcal{E}$ is a subset of $\mathcal{E} \times \mathcal{E}$. It is an *equivalence relation* if it is reflexive, symmetric and transitive. The *equivalence closure of a relation* U is the smallest equivalence relation containing U.

The union-find data type maintains the equivalence closure of a relation $U = \{\ (e_1, e'_1) \ldots (e_p, e'_p)\ \}$ given incrementally (on-line) as a sequence of operations $\mathit{Union}(e_1, e'_1) \ldots \mathit{Union}(e_p, e'_p)$. Each equivalence class is identified by its *representative*, which is a certain element of the class. After initializing the data type with the singleton classes $\{e_1\}, \{e_2\}, \ldots, \{e_n\}$, it supports the operations:

- $\mathit{Union}(e, e')$: merges the classes containing e and e' into a new class. We will assume that e and e' were not in the same class prior to the operation, or, equivalently, that *redundant* unions are ignored.
- $\mathit{Find}(e)$: returns the current representative of the class containing e.

A very well-known implementation of this data type is a set of trees, i.e., a forest, where each tree represents one class. Each node in a tree is (labelled with) some element e_i, and the root of a tree is the representative of that class. Then, $\mathit{Find}(e)$ amounts to returning the root r_e of its tree, and each $\mathit{Union}(e, e')$ first finds the two roots by doing $\mathit{Find}(e)$ and $\mathit{Find}(e')$, and then adds the tree rooted with r_e as an additional child of $r_{e'}$ (or vice versa). This is implemented efficiently by an array A of n integers where $A[i] = j$ if the parent of e_i is e_j, and $A[i] = -1$ if e_i is a root (i.e., a representative). This way, the cost of both operations depends only on the depth of the trees. This depth can be kept logarithmic in n, by adding, in each *Union* operation, the tree with fewer nodes as an additional child of the larger one's root. Then, each time an element increases its depth by one, the size of its class is at least doubled, which cannot happen more than $\log n$ times. Note that the size of each tree can be kept as a negative number at its root. Thus, both operations can be done in $O(\log n)$.

Example 3. Below we show a (numbered) sequence of 12 *Union* operations and the tree and array representations of the resulting two classes. Each edge in the trees is labelled with the union that caused it.

$$\underbrace{(1,8)}_{1}, \underbrace{(7,2)}_{2}, \underbrace{(3,13)}_{3}, \underbrace{(7,1)}_{4}, \underbrace{(6,7)}_{5}, \underbrace{(9,5)}_{6}, \underbrace{(9,3)}_{7}, \underbrace{(14,11)}_{8}, \underbrace{(10,4)}_{9}, \underbrace{(12,9)}_{10}, \underbrace{(4,11)}_{11}, \underbrace{(10,7)}_{12}$$

$$\begin{tikzcd}
 & e_8 & & e_{11} \arrow[ll, "12"'] & & & e_{13} & \\
e_1 \arrow[ur, "1"] & e_2 \arrow[u, "4"] & e_6 \arrow[ul, "5"] & e_4 \arrow[u, "11"] & e_{14} \arrow[ul, "8"] & e_3 \arrow[ur, "3"] & e_5 \arrow[u, "7"] & e_{12} \arrow[ul, "10"] \\
 & e_7 \arrow[u, "2"] & & e_{10} \arrow[u, "9"] & & & e_9 \arrow[u, "6"] &
\end{tikzcd}$$

8	8	13	11	13	8	2	−9	5	4	8	13	−5	11
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*

□

An optimization known as *path compression* aims at further decreasing the trees' depth: at each *Find*(e), for every element e' on the path from e to r_e a direct shortcut to r_e is created, that is, all such e' become children of r_e; this comes at the expense of (roughly) duplicating the work at each *Find*. It turns out (see [Tar75]) that a sequence of $n-1$ *Union* operations (i.e., in the end there is only one class), intermixed with $m \geq n$ *Find* operations, is processed in $\Theta(m\ \alpha(m,n))$ time by the algorithm with path compression, where $\alpha(m,n)$ is a *very* slowly growing function. This $\Theta(m\ \alpha(m,n))$ bound is optimal [Tar79].

Our purpose here is to extend the data structure in order to support the following operation, that is able to explain at any point of the computation "why" two given elements e and e' are equivalent at that moment:

– *Explain*(e,e'): if a sequence U of unions of pairs $(e_1,e'_1)\ldots(e_p,e'_p)$ has taken place, it returns a minimal subset E of U such that (e,e') belongs to the equivalence relation generated by E and it returns $\perp$ if no such E exists.

Example 4 (Example 3 continued). On the previous example, *Explain*(e_1,e_4) returns the explanation $\{(e_7,e_1),(e_{10},e_7),(e_{10},e_4)\}$. □

Proposition 1. *The subset E returned by Explain is unique if it exists.*

The previous property is easy to see by considering the undirected graph which has as edges the pairs in the sequence U of unions. Since U includes no redundant unions, this graph (which we will re-visit in Section 2.2) has no cycles. Therefore, *Explain*(e,e') consists exactly of the edges on the unique path between e and e'.

2.1 Union-find with an $O(k \log n)$ *Explain* operation

In this section a data structure will be developed that supports optimal *Union* and *Find* operations and where *Explain* takes time $O(k \log n)$ for a k-step explanation. The starting point will be the classical union-find forest implementation, without path compression.

Example 5 (Example 3 continued). Consider again *Explain*(e_1, e_4) on:

$$\underbrace{(1,8)}_{1}, \underbrace{(7,2)}_{2}, \underbrace{(3,13)}_{3}, \underbrace{(7,1)}_{4}, \underbrace{(6,7)}_{5}, \underbrace{(9,5)}_{6}, \underbrace{(9,3)}_{7}, \underbrace{(14,11)}_{8}, \underbrace{(10,4)}_{9}, \underbrace{(12,9)}_{10}, \underbrace{(4,11)}_{11}, \underbrace{(10,7)}_{12}$$

$$\begin{tikzcd}
& e_8 & & e_{11} \arrow[ll, "12"'] & & & & e_{13} & \\
e_1 \arrow[ur, "1"] & e_2 \arrow[u, "4"] & e_6 \arrow[ul, "5"'] & e_4 \arrow[u, "11"] & e_{14} \arrow[ul, "8"'] & & e_3 \arrow[ur, "3"] & e_5 \arrow[u, "7"] & e_{12} \arrow[ul, "10"'] \\
& e_7 \arrow[u, "2"] & & e_{10} \arrow[u, "9"] & & & & e_9 \arrow[u, "6"] &
\end{tikzcd}$$

The key observation is the following. Among the three unions (unions #1, #11, and #12) corresponding to the paths from e_1 and from e_4 to their nearest common ancestor e_8, only the one that occurs *last* (union #12) in the sequence U is sure to belong to *Explain*(e_1, e_4). □

In the example we have seen how to find one union (a, b) in *Explain*(e, e'). The remaining unions can be found with two recursive calls *Explain*(e, a) and *Explain*(b, e'), which, as we will see, gives an algorithm for *Explain* of cost $O(k \log n)$. Note however that the recursive calls could also be *Explain*(e, b) and *Explain*(a, e'); in order to know which one of the two situations applies, it is also necessary to distinguish at each edge in which direction the union was applied, i.e., each edge has an *oriented* associated union. In a given sequence U of unions of pairs $(e_1, e'_1) \ldots (e_p, e'_p)$, a union (e_i, e'_i) will be called *newer* than a union (e_j, e'_j) whenever $i > j$. Now, another technical detail is that the pair (a, b) found as in the example is in fact the newest union in *Explain*(e, e'); therefore, the newest union in both recursive calls will be strictly older than (a, b) and hence an infinite recursion cannot occur. Below all this is formalized.

Lemma 1. *Consider a union-find forest data structure without path compression. For each pair of constants (e, e') with nearest common ancestor c, the newest associated union (a, b) of the paths from e to c and from e' to c belongs to Explain(e, e'). Moreover, (a, b) is the newest union in Explain(e, e').*

The previous lemma can be used in a data structure where the unions are kept as a numbered sequence of (oriented) pairs, and where the array representation now also contains the associated unions corresponding to each edge. Since path compression is necessary in order to have optimal *Find* and *Union* operations, we will also keep, with each element, the parents on the data structure with path compression.

Example 6 (Example 3 continued). For our example, the data structures are:

$$\underbrace{(1,8)}_{1},\ \underbrace{(7,2)}_{2},\ \underbrace{(3,13)}_{3},\ \underbrace{(7,1)}_{4},\ \underbrace{(6,7)}_{5},\ \underbrace{(9,5)}_{6},\ \underbrace{(9,3)}_{7},\ \underbrace{(14,11)}_{8},\ \underbrace{(10,4)}_{9},\ \underbrace{(12,9)}_{10},\ \underbrace{(4,11)}_{11},\ \underbrace{(10,7)}_{12}$$

associated union:	1	4	3	11	7	5	2	−1	6	9	12	10	−1	8
parent w/ path c. :	8	8	13	11	13	8	8	−9	13	11	8	13	−5	11
parent wo/ path c.:	8	8	13	11	13	8	2	−9	5	4	8	13	−5	11
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*

□

Theorem 1. *The previous data structure performs a sequence of $m \geq n$ finds and $n{-}1$ intermixed unions in time $\Theta(m\ \alpha(m,n))$. Moreover, any Explain(e,e') is supported in $O(k \log n)$ where k is the size of the proof.*

Proof. If always *Find* and *Union* are computed first on the compressed trees, in the same way as it is done in [Tar75], the remaining work at each *Union* (updating the non-compressed parents and the associated union) only needs constant time. Hence, this extra work will not affect the $\Theta(m\ \alpha(m,n))$ runtime. For *Explain*(e,e'), by Lemma 1 we can identify one pair of the proof in time $O(\log n)$ from the non-compressed tree which has depth $O(\log n)$. This work will only be repeated k times, and hence, the total complexity for *Explain*(e,e') is $O(k \log n)$. □

2.2 Union-find with an $O(k)$ *Explain* operation

Here we develop a data structure in which *Explain* can be answered in optimal time $O(k)$ for a k-step proof, at the expense of slightly more costly *Union*s, which have an amortized time bound of $O(\log n)$.

The main idea is to consider again, as we did for Proposition 1, the graph which has as edges the pairs in the sequence U of unions. As said, since U includes no redundant unions, this graph has no cycles, i.e., it is a forest, and therefore *Explain*(e,e') consists exactly of the edges on the unique path between e and e'. Of course this forest can be maintained with only constant work at each *Union*, and hence the only problem is how to efficiently find this unique path for a given *Explain* operation.

For this purpose we will choose a root for each tree and direct all its edges towards that root. With this structure being invariant, *Explain*(e,e') will amount to returning the edges in the paths from e and e' to their common ancestor, which is computable in time $O(k)$, k being the length of the proof. This concrete structure, which in the following will be called *proof forest*, can be kept invariant as follows. At each *Union*(e,e'), assume, w.l.o.g., that the tree of e has no more elements than the one of e', and do:

1. Reverse all edges on the path between e and the root of its tree.
2. Add an edge $e \to e'$.

It is not difficult to see that this preserves the aforementioned tree structure, as well as the invariant that the path between two nodes is found by computing their nearest common ancestor. Moreover, each time an edge is reversed, the size of its tree is at least doubled. Therefore we have the following:

Lemma 2. *In a sequence of n–1 Union operations, each edge in the proof forest is reoriented at most $O(\log n)$ times.*

Example 7. (Example 3 revisited).
Assume that again the following sequence of unions takes place:

$$\underbrace{(1,8)}_{1}, \underbrace{(7,2)}_{2}, \underbrace{(3,13)}_{3}, \underbrace{(7,1)}_{4}, \underbrace{(6,7)}_{5}, \underbrace{(9,5)}_{6}, \underbrace{(9,3)}_{7}, \underbrace{(14,11)}_{8}, \underbrace{(10,4)}_{9}, \underbrace{(12,9)}_{10}, \underbrace{(4,11)}_{11}, \underbrace{(10,7)}_{12}$$

Then the proof forest could be as follows (but note that it is not unique):

$$\begin{tikzcd}
 & & 8 \arrow[r] & 1 & 7 & 2 \arrow[l] & 12 \arrow[r] & 9 \arrow[r] & 3 \arrow[r] & 13 \\
14 \arrow[r] & 11 \arrow[r] & 4 \arrow[r] & 10 \arrow[ur] & 6 \arrow[u] & & & 5 \arrow[u] & &
\end{tikzcd}$$

$$1 \rightarrow 7$$

□

The algorithm we propose is to use the standard union-find with path compression and maintain at the same time the proof forest, which can be represented by an array of pointers (integers) to parents, as it is done in the union-find data structure itself. Altogether, the only operation whose cost will be increased is *Union*.

Theorem 2. *In a sequence of $m \geq n$ finds and $n-1$ intermixed unions, the previous data structure performes each Union in an amortized time bound of $O(\log n)$. Moreover, any Explain(e, e') operation is supported in $O(k)$ where k is the size of the proof.*

Proof. For every call to *Union* the only extra work to be done is the reorientation of the appropriate edges. Since we will have a maximum of $n-1$ edges and each edge will be reoriented at most $O(\log n)$ times, this extra work will be $O(n \log n)$ in the whole sequence, hence giving an amortized time bound of $O(\log n)$ for each *Union*. Note that the *Find* operations are still as efficient as in [Tar75].

As explained above, *Explain*(e, e') will consist of the edges in the paths from e and e' to their common ancestor, which is computable in time $O(k)$ with the invariant structure of the proof forest. □

3 Incremental congruence closure with *Explain*

Let $\mathcal{F}$ be a set of (fixed-arity) function symbols and let $T(\mathcal{F})$ be the set of terms without variables built over $\mathcal{F}$: all constants (0-ary symbols) are terms, and $f(t_1, \ldots, t_n)$ is a term whenever f is a non-constant n-ary symbol and $t_1, \ldots, t_n$ are terms. A binary relation $=$ over $T(\mathcal{F})$ is a *congruence relation* if it is reflexive, symmetric, transitive and monotonic; the latter property states, for every non-constant function symbol f, that $f(x_1 \ldots x_n)=f(y_1 \ldots y_n)$ whenever $x_i=y_i$ for all i in $1 \ldots n$. The *congruence closure of a relation* U is the smallest congruence relation containing U. Well-known algorithms for computing the congruence closure of a given set of equations between terms without variables were already given in the early 1980s, such as the $O(n \log n)$ DST algorithm by Downey, Sethi, and Tarjan, [DST80], the Nelson-Oppen one of [NO80] and Shostak's algorithm [Sho78].

Here we will define an *incremental* congruence closure algorithm: we consider a sequence of n *Merge*(s,t) operations, for terms s and t, intermixed with *AreCongruent?*(s,t) operations asking whether s and t are currently congruent, and *Explain*(s,t) operations for recovering the original *Merge* operations causing s and t to be congruent.

3.1 Initial assumptions and operations

We will use as a starting point the (non-incremental) congruence closure algorithm of [NO03], which is essentially a simplification of the DST algorithm. DST needs an initial transformation to directed acyclic graphs of outdegree 2, which in [NO03] is replaced by another one, at the formula representation level. This is done by *Currifying*, as in the implementation of functional languages; as a result, there will be only one binary "apply" function symbol (denoted here by an f) and constants. For example, Currifying $g(a, h(b), b)$ gives $f(f(f(g,a), f(h,b)), b)$.

Furthermore, as in the abstract congruence closure approaches (see [Kap97], [BT00]), new constant symbols c are introduced for giving names to non-constant proper subterms t; such t are then replaced everywhere by c, and the equation $t{=}c$ is added. Then, in combination with Currification, one can obtain the same efficiency as in more sophisticated DAG implementations by appropriately indexing the new constants such as c, which play the role of the pointers to the (shared) subterms such as t in the DAG approaches. For example, the equation $f(f(f(g,a), f(h,b)), b)=b$ is flattened by replacing it by the four equations $f(g,a)=c$, $f(h,b)=d$, $f(c,d)=e$, and $f(e,b)=b$.

These two (structure-preserving) transformations can be done in linear time, and, *in all practical applications we are aware of*, also *once and for all*, instead of at each call to the congruence closure procedure. The transformations could even be done back-and-forth at each operation without increasing the asymptotic complexity bounds (although then the k in the complexity of *Explain* becomes the proof *size*, rather than its number of steps, since each step can involve large terms).

Hence, along this section, we will assume that the equations input to *Merge* are of the form $f(a,b)=c$, or of the form $a=b$, where a, b and c always denote constants. This makes the algorithm surprisingly simple and clean and more efficiently implementable than algorithms for arbitrary terms (in fact, its non-incremental version of [NO03] is about 50 times faster than other recent implementations such as [TV01] on the benchmarks of [TV01]). Due to its simplicity, our algorithm is also easier to extend; e.g., in [NO03] we considered congruence closure with *integer offsets*.

In the remainder of this paper we consider an abstract data type for incremental congruence closure with the following operations:

– $\mathit{Merge}(t, c)$: where t is a flat term of the form $f(a, b)$ or a constant a.
– $\mathit{AreCongruent?}(s, t)$: returns "*yes*" if s and t currently belong to the same congruence class and "*no*" otherwise.
– $\mathit{Explain}(s, t)$: assume a sequence M of merges $(s_1, t_1) \ldots (s_p, t_p)$ has occurred, and that (s, t) is in the congruence closure of M; then $\mathit{Explain}(s, t)$ returns a subset $E = (s_{i_1}, t_{i_1}) \ldots (s_{i_k}, t_{i_k})$ of M, with $1 \leq i_1 < \ldots < i_k \leq p$, such that exactly at the i_k-th merge s and t became congruent, due to the merge operations in E.

Theorem 3. *A sequence of n Merge operations can be processed in $O(n \log n)$ time, and hence each one of them in $O(\log n)$ amortized time. Furthermore, each question AreCongruent?(s, t) can be answered in $O(|s| + |t|)$, i.e., in constant time if s and t are constants. For the Explain(s, t) operation between constants, a k-step proof can be found in time $O(k\,\alpha(k, k))$, to which, for arbitrary terms s and t, an additional cost $O(|s| + |t|)$ has to be added.*

3.2 Implementation of *Merge*

The underlying union-find data structure used here applies *eager path compression*, that is, for each constant symbol c_i, its representative can always be returned in constant time by accessing an array *Representative*$[i]$. In order to maintain this *Representative* table, there will be additional *Class Lists* containing for each representative the constants in its class.

The basic data structures for the congruence closure algorithm are:

1. *Pending*: a list whose elements are input equations $a{=}b$, or pairs of input equations $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$ where a_i and b_i are already congruent for $i = 1, 2$. In both cases, when inserting such an element in *Pending*, what is pending is the merge of the constants a and b.
2. The *Use lists*: for each representative a, $\mathit{UseList}(a)$ is a list of input equations $f(b_1, b_2){=}b$ such that a is the representative of b_1 or of b_2 (or of both).
3. The *Lookup table*: for all pairs of representatives (b, c), $\mathit{Lookup}(b, c)$ is some input equation $f(a_1, a_2){=}a$ such that b and c are the current respective representatives of a_1 and a_2 iff such an equation exists. Otherwise, $\mathit{Lookup}(b, c)$ is $\perp$. A useful additional invariant is that, for representatives b and c, $f(a_1, a_2) = a$ is in $\mathit{UseList}(b)$ and in $\mathit{UseList}(c)$ iff $\mathit{Lookup}(b, c)$ is $f(a_1, a_2) = a$.
4. The *Proof forest* data structure, as presented in the previous section.

Here we present the algorithms in a way as similar as possible to the non-incremental one of [NO03], and separate the treatment of the proof forest from the congruence closure algorithm itself. The data structures are initialized as expected: all *Use Lists*, *Pending*, and the *Proof forest* are empty, and $\mathit{Lookup}(a, b)$

is $\perp$ for all pairs (a, b). Each *ClassList*(a) is initialized to contain only a and each *Representative*(a) is initialized with a. Note that *Lookup* could also be stored in a hash table (since a 2-dimensional array will be almost empty), and that the non-constant time initializations can also be avoided[1]. In the following algorithms, a' always denotes *Representative*(a) for each constant a.

```
 1. Procedure Merge($s$=$t$)
 2.   If $s$ and $t$ are constants $a$ and $b$ Then {
 3.     add $a$=$b$ to Pending
 4.     Propagate() }
 5.   Else /* $s$=$t$ is of the form $f(a_1, a_2)$=$a$ */
 6.     If Lookup($a'_1, a'_2$) is some $f(b_1, b_2)$=$b$ Then {
 7.       add ( $f(a_1, a_2)$=$a$, $f(b_1, b_2)$=$b$ ) to Pending
 8.       Propagate() }
 9.     Else {
10.       set Lookup($a'_1, a'_2$) to $f(a_1, a_2)$=$a$
11.       add $f(a_1, a_2)$=$a$ to UseList($a'_1$) and to UseList($a'_2$) }

12. Procedure Propagate()
13.   While Pending is non-empty Do {
14.     Remove $E$ of the form $a$=$b$ or ($f(a_1, a_2)$=$a$, $f(b_1, b_2)$=$b$) from Pending
15.     If $a' \neq b'$ and, wlog., |ClassList($a'$)| $\leq$ |ClassList($b'$)| Then {
16.       old_repr_a := $a'$
17.       Insert edge $a \rightarrow b$ labelled with $E$ into the proof forest
18.       For each $c$ in ClassList(old_repr_a) Do {
19.         set Representative($c$) to $b'$
20.         move $c$ from ClassList(old_repr_a) to ClassList($b'$) }
21.       For each $f(c_1, c_2)$=$c$ in UseList(old_repr_a) Do
22.         If Lookup($c'_1, c'_2$) is some $f(d_1, d_2)$=$d$ Then {
23.           add ($f(c_1, c_2)$=$c$, $f(d_1, d_2)$=$d$) to Pending
24.           remove $f(c_1, c_2)$=$c$ from UseList(old_repr_a) }
25.         Else {
26.           set Lookup($c'_1, c'_2$) to $f(c_1, c_2)$=$c$
27.           move $f(c_1, c_2)$=$c$ from UseList(old_repr_a) to UseList($b'$) }}}
```

Each iteration of the *Propagate()* algorithm picks a pending union. If this union is not redundant, it is added to the proof forest (line 17) and (lines 19 and 20) to the union-find data structure. Lines 21–27 traverse the *UseList* of the constant whose representative has changed and, checking the lookup table, detect new pairs of constants to be merged.

[1] For example, *Representative*$[a]$ can be updated by storing in *Representative*$[a]$ an index k to an auxiliary array A, where $A[k]$ contains a and its representative, and with a counter max indicating that, for all $k < max$, $A[k]$ contains correct (i.e., initialized) information; re-initialization then simply amounts to setting max to 0 (in fact, this general idea can always avoid any n-dimensional array initialization).

3.3 Complexity of *Merge* and *AreCongruent?*

As said, an amortized analysis is done over the whole sequence of n *Merge* operations. The procedure *Merge* itself has no loops. Concerning *Propagate()*, let m be the number of different constants (note that $m \leq 3n$). The loop at lines 19 and 20 is executed in total $O(m \log m)$ times, namely when some constant changes its representative, which for each one of the m constants happens at most $\log m$ times, because each time the size of its class is at least doubled. Line 17 inserts an edge between pair of constants in the proof forest. This is done at most m–1 times, and by Lemma 2 the total time for the m–1 insertions is $O(m \log m)$. In the loop at lines 21–27, each one of the at most n input equations of the form $f(c_1, c_2){=}c$ is treated when c_1 or c_2 changes its representative (which, as before, cannot happen more than $\log m$ times). Altogether, we obtain an $O(n \log n)$ runtime. Re-using *UseList* and *ClassList* nodes, only linear space is required.

Note that the equivalence relation between all constants is dealt with by the *Representative* array, i.e., by union-find with "eager path compression". This makes the algorithm simpler and allows one to handle *AreCongruent?*(a, b) in constant time; asymptotically speaking, it causes no overhead to *Merge*. In practice, lazy path compression may globally perform better since strictly less "compressions" are done, although it has an overhead caused by additional checks (whether the representative has been reached or not, etc.). In any case, all asymptotic bounds given here carry over straightforwardly to the case of lazy path compression.

3.4 Implementation of *Explain*

As said, each edge $a - b$ in the proof forest is labelled with a single input equation $a{=}b$ or with a pair of input equations $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$. The way the proof forest (and the information associated to its edges) is represented is not described here; it can be done e.g., as in Subsection 2.2, by an array of pointers.

Example 8. Below we show a (numbered) sequence of 6 *Merge* operations and the state of the proof forest after processing them. Each edge of the proof forest is annotated with its corresponding input equation or pair of input equations:

$$\underbrace{f(g,h){=}d}_{1},\ \underbrace{c{=}d}_{2},\ \underbrace{f(g,d){=}a}_{3},\ \underbrace{e{=}c}_{4},\ \underbrace{e{=}b}_{5},\ \underbrace{b{=}h}_{6}, \qquad a \xrightarrow{1,3} d \xleftarrow{2} c \xleftarrow{4} e \xleftarrow{5} b \xleftarrow{6} h$$

On an *Explain*(a, b) operation, the nearest common ancestor d is detected, and the merge operations on the paths $a - d$ (1,3) and $b - d$ (5,4,2) are output as part of the proof; but from 1 and 3 also recursively *Explain*(h, d) needs to be output. In order to obtain the desired complexity bound, it is necessary to avoid repeated visits to nodes like b, e, c, d in such recursive calls. After the merge operations in the path $b - d$ have been output, the constants b, e, c and d can be considered to be inside the same equivalence class C. Since the information in the edges in the path $b - d$ has already been output, in any future traversal one can jump from any element of C to d (here d is the *highest node* of C, the element of C that is closest to the root of its tree in the proof forest). Hence, in the recursive call to *Explain*(h, d), only the edge $b - h$ is traversed, since from b one can directly jump to d. □

The data structures for avoiding such repeated visits and the way they are used is explained in the following two points:

The additional union-find, and *HighestNode*. At each call to *Explain*(s, t), an additional union-find data structure with path compression keeps track of the classes of constants that are already equivalent by the proof output so far. More precisely, apart from the *Find*(a) operation, there is also a *HighestNode(a)* operation, which returns the *highest node* among all nodes of the proof tree in the equivalence class of a; this highest node is simply stored at the node of *Find*(a). Maintaining the *HighestNode* information will be easy: since only unions of the form *Union*$(a, parent(a))$ take place, the *HighestNode* of the new class is always the *HighestNode* of the second argument of the call, i.e. the *HighestNode* of $parent(a)$.

Finding the nearest common ancestor in the proof forest. There is also a *NearestCommonAncestor(a,b)* operation that retrieves the *highest node* of the class of the nearest common ancestor of a and b in the proof forest. When looking for it, as it happens in the *ExplainAlongPath* procedure below, one has to jump over whole classes of equivalent constants by means of the *HighestNode* operation in order to avoid traversing unnecessary edges.

Now we are ready to present the two procedures implementing *Explain*:

```
1.  Procedure Explain($c_1, c_2$)
2.    Set PendingProofs to $\{c_1=c_2\}$
3.    While PendingProofs is not empty Do {
4.      Remove an equation $a=b$ from PendingProofs
5.      $c$ := NearestCommonAncestor(a,b)
6.      ExplainAlongPath(a,c)
7.      ExplainAlongPath(b,c) }

8.  Procedure ExplainAlongPath(a,c)
9.    $a$ := HighestNode(a)
10.   While $a \neq c$ Do {
11.     $b$ := $parent(a)$
12.     If edge $a \rightarrow b$ is labelled with a single input merge $a=b$
13.       Output $a=b$
14.     Else {   /* edge labelled with $f(a_1,a_2)=a$ and $f(b_1,b_2)=b$ */
15.       Output $f(a_1,a_2)=a$ and $f(b_1,b_2)=b$
16.       Add $a_1=b_1$ and $a_2=b_2$ to PendingProofs }
17.     Union$(a,b)$
18.     $a$ := HighestNode(b) }
```

3.5 Complexity of *Explain*

Let k be the number of steps in the final proof that is output. There are in total $O(k)$ iterations of the *ExplainAlongPath* loop since at each iteration either one (line 13) or two (line 15) such steps are output. In fact, for each call of the form *ExplainAlongPath(a,c)*, the number of iterations corresponds to the number of different equivalence classes along the path from a to c, and at each iteration, one union between classes takes place, as well as one call to *HighestNode* (i.e., one *Find*). Hence in total $O(k)$ such classes are merged along the whole proof. The total work done for searching nearest common ancestors (line 5 of procedure *Explain*) is also $O(k)$, because it can be done in time linear in the number of classes that are merged in the subsequent two calls *ExplainAlongPath(a,c)* and *ExplainAlongPath(b,c)*. Furthermore, for each iteration of *ExplainAlongPath*, at most two equalities are added to *PendingProofs*, and hence the loop of procedure *Explain* is executed $O(k)$ times. Altogether, the global runtime is dominated by the $O(k)$ unions of classes and the $O(k)$ calls to *Find*, which has a total cost of $O(k\ \alpha(k,k))$ in the union-find algorithm with path compression.

3.6 Quality of explanations. Experiments.

Example 9. After a given sequence of input equations E, there can be several explanations for an equation $s{=}t$. Consider the sequence of 7 input equations E:

$a{=}b_1 \quad b_1{=}b_2 \quad b_2{=}b_3 \quad b_3{=}c \quad f(a_1,a_1){=}a \quad f(c_1,c_1){=}c \quad a_1{=}c_1$

In our algorithm, *Explain*($a{=}c$) will return the first four equations, although the last three equations also form a correct explanation of $a{=}c$. □

Finding *short* explanations is good for most practical applications, and also finding the *oldest* explanation (i.e., the one contained in the shortest prefix of the sequence E) is desirable (roughly, because it allows one to do more powerful backjumping). Since our algorithm always returns the oldest explanation (see the definition of *Explain* before Theorem 3), from now on we will focus on length.

Unfortunately, trying to always find the *shortest* explanation (in number of steps) is too ambitious: given such an E, an equation $s{=}t$, and a natural number k, deciding whether an explanation of size smaller than k exists for $s = t$ is already an NP-hard problem[2]. Therefore, the usual criterion for quality of an explanation is its *irredundancy*: after removing any step, it is no longer a valid explanation. Surprisingly, the explanations found by our algorithm as presented in the previous subsection sometimes still contain redundant steps.

Example 10. After the sequence of input equations:

$a_1{=}b_1 \quad a_1{=}c_1 \quad f(a_1,a_1){=}a \quad f(b_1,b_1){=}b \quad f(c_1,c_1){=}c$

the proof forest may consist of the two trees: $a \rightarrow b \leftarrow c$ and $b_1 \rightarrow a_1 \leftarrow c_1$. Now *Explain*($a = c$) will return all five equations. However, the two equations containing b_1 are redundant. □

[2] Ashish Tiwari. Personal communication.

We have run our algorithm as given in the previous subsection over a very large set of benchmarks (all the EUF examples mentioned in [GHN+04], available at the second author's home page). There, on average, explanations have 12.6 steps; redundant explanations are returned in 13.56 percent of the cases, having, on average, 40 steps of which 6 are redundant.

Fortunately, one can easily and efficiently post-process explanations in order to fully remove all redundant steps. On the one hand, it is not very hard to see that one of our explanations can be redundant only if it contains at least three equations of the same *structural class*, i.e., of the form $f(a_1, a_2){=}a,\ \ f(b_1, b_2){=}b,\ \ f(c_1, c_2){=}c$ where a_i, b_i and c_i have the same representative for $i = 1, 2$. This can be checked in time linear in k, and is an extremely good filter: three such equations occur only in 0.27 percent of the irredundant explanations.

The 13.8 percent of the explanations marked as "possibly redundant" by this test can be post-processed as follows in time $O(k^2 \log k)$ in order to remove all redundancies: while not all equations are marked as "necessary", pick an unmarked one, remove it if the remaining equations are still a correct explanation (checking this takes $O(k \log k)$ time), and otherwise mark it as "necessary".

Proof forests with structural classes as nodes. We have also implemented a variant of our proof forests where the nodes are these structural classes and hence all edges are labelled with a single input equation between constants. Now, instead of inserting edges labelled with $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$, one merges the two nodes (classes) $[...a...]$ and $[...b...]$ into a single one.

Example 11. Consider again the input sequence of the previous example:
$a_1{=}b_1 \quad a_1{=}c_1 \quad f(a_1, a_1){=}a \quad f(b_1, b_1){=}b \quad f(c_1, c_1){=}c$
Now the proof forest will consist of the two trees: $[a, b, c]$ and $b_1 \rightarrow a_1 \leftarrow c_1$ and *Explain*$(a{=}c)$ will return only the structural equations involving a and c and its corresponding recursive explanation that $a_1{=}c_1$. □

In such proof forests, the *Explain* operation is implemented in a very similar way as before. For simplicity, in the previous subsection we have not mentioned this improvement, but it is not hard to see that all results apply.

With this new approach, only 3.5 percent of the explanations are still redundant, having on average 34 steps, of which 6 are redundant. Using the test, now postprocessing is needed only in 3.95 percent of the cases.

Example 12. Let's see why some redundancies can still appear. Consider:
1. $f(a_1, a_1){=}a$ 2. $f(b_1, b_1){=}b$ 3. $f(c_1, c_1){=}c$ 4. $f(d_1, d_1){=}d$
5. $a_1{=}b_1$ 6. $c_1{=}d_1$ 7. $a_1{=}c$ 8. $a_1{=}a$ 9. $d{=}d_1$

Then the proof tree may become:
$$\begin{tikzcd}
{[a,b]} \arrow[r] & a_1 & {[c,d]} \arrow[l] & d_1 \arrow[l] & c_1 \arrow[l] \\
 & b_1 \arrow[u] & & &
\end{tikzcd}$$

and *Explain*$(b{=}d_1)$ returns the set of all 9 input equations, of which #1 and #8 are redundant. This redundancy is caused by the two equivalent classes $[a, b]$ and $[c, d]$. Indeed, it can be shown that if no two such equivalent non-singleton structural classes exist, proofs will always be irredundant. But it seems too expensive to maintain that property during the congruence closure procedure; in particular, the difficulties arise when two such classes become equivalent (in the example, after $d{=}d_1$) while they are already in the same tree, i.e., when they are already equal by equations between constants. □

4 Related and Future Work

To our knowledge, this is the first congruence closure algorithm able to produce explanations in time that does not depend on the number of input equations n. Moreover, the congruence closure algorithm itself is not only simple, but it also runs in the best known time, namely $O(n \log n)$, and is indeed very fast in practice.

We believe that this kind of fundamental algorithmic developments are extremely useful, because we have seen several less adequate ad-hoc solutions being applied in modern deduction and verification tools. E.g., in [BDS02] where the CVC tool is described (`verify.stanford.edu/CVC`), a trial-and-error method for finding explanations is given. Another example of this phenomenon is SRI's "lemmas-on-demand" approach in the ICS tool: in [dMR02] it is mentioned that "Unfortunately, current domain-specific decision procedures lack such a conflict explanation facility. Therefore, we developed an algorithm that calls C-solver $O(k \times n)$ times, where k is given, for finding such an overapproximation". Several authors have attacked the specific problem of generating explanations in the context of union-find and congruence closure [SD99,KS04,dMRS04]). In particular, in the paper "Justifying Equality" [dMRS04], for union-find *Explain* is done in time $O(n\,\alpha(n))$, i.e., it depends on the number of unions that have taken place. For the (strict) generalization to congruence closure, this is indeed also the case (although no concrete bound is given in that paper), and the notion of *local irredundancy* achieved in [dMRS04] already holds for our basic algorithm of Section 3.

Concerning future work, we plan to study extensions, such as the version for congruence closure with *integer offsets* [NO03], to which the ideas given here for *Explain* can also be applied. It also remains to be studied whether irredudant proofs can be generated directly without any postprocessing (although this does not seem to lead to more practical efficiency).

References

[ABC$^+$02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE-18*, LNCS 2392, pages 195–210, 2002.

[BDS02] Clarke Barrett, David Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation into sat. In *Procs. 14th Intl. Conf. on Computer Aided Verification (CAV)*, LNCS 2404, 2002.

[BT00] L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Conf. Autom. Deduction, CADE*, LNAI 1831, pages 64–78, 2000.

[CLR90] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[dMR02] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Procs. 5th Int. Symp. on the Theory and Applications of Satisfiability Testing, SAT'02*, pages 244–251, 2002.

[dMRS04] L. de Moura, H. Rueß, and N. Shankar. Justifying equality. In *Proc. of the Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Cork, Ireland, 2004.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *J. of the Association for Computing Machinery*, 27(4):758–771, 1980.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explanation. In *Procs. 15th Int. Conf. on Computer Aided Verification (CAV)*, LNCS 2725, 2003.

[GHN$^+$04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In *Procs. 8th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1232, 1997.

[KS04] Robert Klapper and Aaron Stump. Validated proof-producing decision procedures. In *Proceedings of the Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Cork, Ireland, 2004.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC'01)*, 2001.

[NO80] 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.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In *10h Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, LNAI 2850, pages 78–90, 2003.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, 1999.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7), 1978.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, April 1975.

[Tar79] Robert Endre Tarjan. A class of algorithms that require nonlinear time to maintain disjoint sets. *J. Comput. and Sys. Sci.*, 18(2):110–127, 1979.

[TV01] Ashish Tiwari and Laurent Vigneron. Implementation of Abstract Congruence Closure with randomly generated CC problem instances, 2001. At www.csl.sri.com/users/tiwari.