Fast Congruence Closure and Extensions

Robert Nieuwenhuis [1,3,*] , Albert Oliveras [1,2]

Technical University of Catalonia, Dept. LSI,
Jordi Girona 1, 08034 Barcelona, Spain

Abstract

Congruence closure algorithms for deduction in ground equational theories are ubiquitous in many (semi-)decision procedures used for verification and automated deduction. In many of these applications one needs an *incremental* algorithm that is moreover capable of recovering, among the thousands of input equations, the small subset that *explains* the equivalence of a given pair of terms.

In this paper we present an algorithm satisfying all these requirements. First, building on ideas from abstract congruence closure algorithms [Kapur (1997,RTA), Bachmair & Tiwari (2000,CADE)], we present a very simple and clean incremental congruence closure algorithm and show that it runs in the best known time $O(n \; \log n)$. After that, we introduce a proof-producing union-find data structure that is then used for extending our congruence closure algorithm, without increasing the overall $O(n \; \log n)$ time, in order to produce a k-step explanation for a given equation in almost optimal time (quasi-linear in k).

Finally, we show that the previous algorithms can be smoothly extended, while still obtaining the same asymptotic time bounds, in order to support the interpreted functions symbols *successor* and *predecessor*, which have been shown to be very useful in applications such as microprocessor verification.

Key words: decision procedures, congruence closure, equational reasoning, verification
PACS: 89.20.Ff, 84.30.Bv, 07.05.Bx, 95.75.Pq, 02.10.-v

---

[*] Corresponding author.
URLs: `www.lsi.upc.edu/~roberto` (Robert Nieuwenhuis),
`www.lsi.upc.edu/~oliveras` (Albert Oliveras).
[1] Partially supported by Spanish Min. of Educ. and Science through the LogicTools project (TIN2004-03382).
[2] Partially supported by Spanish Min. of Educ. and Science through a FPU grant AP2002-3533.
[3] Partially supported by personal grant "SMT Solvers for High-Level Hardware Verification" from Intel Corporation.

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 $f(a_1 \dots a_n) = f(b_1 \dots b_n)$ whenever $a_i = b_i$ for all i in $1 \dots n$.

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.* □

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 the case in decision procedures for SMT (SAT Modulo Theories), i.e., procedures for deciding the satisfiability of Boolean formulae over ground atoms with symbols that are interpreted in some theory. One important class of SMT problems is called EUF (Equality with Uninterpreted Functions), containing atoms that are equalities between terms built over uninterpreted function symbols. EUF (i.e., SAT modulo the theory of congruences) is important in applications such as the verification of pipelined processors [1], where, if the control is verified, the concrete data operations can be abstracted by uninterpreted function symbols.

The *lazy* approaches to SMT, e.g. [2–6] 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 propositional SAT solver. If the SAT solver reaches a (partial) model that is not a congruence, the model is precluded by adding a new propositional clause called an explanation; 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 the new clause $b = d \wedge f(b) = d \wedge f(d) = a \quad \longrightarrow \quad a = b$, 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) input equations.* □

Another recent approach for the flexible generation of decision procedures is called DPLL(T) [7]. 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 [8,9]. 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 [10]. 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 [7] for details about the DPLL(T) approach and experiments on benchmarks from a large variety of verification problems). In the 2005 SAT Modulo Theories Competition (SMT-COMP; see [11] as well as the results on the web), our DPLL(T) implementation in BarcelogicTools won all four divisions in which it participated (out of seven divisions).

Since in such an incremental setting many *Explain* operations occur during a single congruence closure procedure, it is important to efficiently recover these explanations, even at the expense of making the congruence closure algorithm slightly slower in practice. Recovering small explanations is well-known to be crucial for efficiency, but one cannot afford to use a naive approach for that, since then the cost of *Explain* would strongly 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, quasi-linear in the size of the explanation, without increasing the asymptotic $O(n \log n)$ runtime of the overall congruence closure.

This article is structured as follows.

Some preliminaries on terms, relations, congruences and on the classical Union-Find data structure are given in Section 2.

In Section 3 we describe a simple and efficient algorithm for incremental congruence closure, whose implementation is described in Subsection 3.3. Despite an expert reader may skip Subsections 3.4 and 3.5, we include them here for the self-containedness of the paper. In them, we prove the correctness of the algorithm and we show that it runs in $O(n \log n)$ time, as the fastest known congruence closure algorithms.

Section 4 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 describe a 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 $O(\log n)$.

In Section 5 the latter union-find data structure is applied inside our incremental congruence closure algorithm, in order to produce explanations. Its complexity is analyzed in Subsection 5.1, 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. The *Explain* operation is given in Subsection 5.2 and analyzed in detail in Subsection 5.3 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 5.4 discusses quality issues of explanations, gives extensive experimental results, and introduces several extensions with practical impact of our explanation algorithms.

In Section 6 the whole framework is extended in order allow an integer interpretation and input formulae containing arbitrary terms built over uninterpreted symbols as well as over the interpreted functions symbols *successor* and *predecessor*. Equivalently, one can consider that all subterms t can in fact be of the form $t + k$, for some concrete integer value k, called an *integer offset*. Dealing with EUF with integer offsets has been shown to be very useful in applications such as microprocessor verification [12]. Here we prove that with this extension of our algorithms the same asymptotic time bounds can be maintained for all operations.

We finally compare with related work and conclude in Section 7.

2 Preliminaries

Let $\mathcal{F}$ be a finite set of function symbols with an arity function arity: $\mathcal{F} \to \mathbb{N}$. Function symbols g with $arity(g) = n$ are called *n-ary* symbols (when $n = 1$, one says *unary* and when $n = 2$, *binary*). If $arity(g) = 0$, then g is a *constant symbol*. The set of ground terms over $\mathcal{F}$, denoted by $\mathcal{T}(F)$, is the smallest set containing all constant symbols such that $g(t_1, \ldots, t_n)$ is in $\mathcal{T}(F)$ whenever $g \in \mathcal{F}$, $arity(g) = n$, and $t_1, \ldots, t_n \in \mathcal{T}(F)$. In the rest of the paper, possibly subscripted or primed a, b, c, d and e always denote constants. Similarly, s, t and u will denote arbitrary terms and f, g and h denote non-constant function symbols.

By $|s|$ we denote the *size* (number of symbols) of a ground term s: we have $|a| = 1$ if a is a constant symbol and for non-constant terms we have that $|g(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n|$. The *depth* of a term s is denoted by $depth(s)$

and is defined: $depth(a) = 1$ if a is a constant symbol and $depth(g(t_1, \ldots, t_n)) = 1 + max(depth(t_1), \ldots, depth(t_n))$.

A binary *relation* R 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. An equivalence relation induces a partition of $\mathcal{E}$ into *equivalence classes*. Two elements a and b belong to the same equivalence class if and only if $(a, b) \in R$. In this case, we will say that they are *equivalent* (in R). Given a binary relation R, its *equivalence closure* is the smallest equivalence relation containing R.

A binary relation R on $T(\mathcal{F})$ is *monotonic* if $(g(s_1, \ldots, s_n)\,,\, g(t_1, \ldots, t_n)\,) \in R$ whenever g is an n-ary function symbol in $\mathcal{F}$ and $(s_i\,,\, t_i) \in R$ for all i in $1 \ldots n$. A *congruence relation* is a monotonic equivalence relation. Any congruence relation R induces a partition of $T(\mathcal{F})$ into *congruence classes*, where two terms s and t belong to the same congruence class if and only if $(s, t) \in R$. In this case we say that s and t are *congruent* (in R). A (ground)*equation* is an (unordered) pair of ground terms (s, t), denoted by $s = t$. The size of an equation $s = t$ will be $1 + |s| + |t|$ and the size of a set of equations E, denoted $|E|$, will be the sum of the sizes of its equations. Given a set of equations E built over $\mathcal{F}$, we denote by E^* the *congruence closure* of E: the smallest congruence relation over $T(\mathcal{F})$ containing E. We sometimes write $E \models s = t$ to denote that (s, t) belongs to E^*; if E' is a set of equations, we write $E \models E'$ to denote that $E \models s = t$ for all $s = t$ in E', and we write $E \equiv E'$ to denote that $E \models E'$ and $E' \models E$.

Here we also give some basic notions about the union-find data structure (see, e.g., [13] for details). The union-find data type maintains the equivalence closure of a binary relation $U = \{\ (e_1, e'_1), \ldots, (e_p, e'_p)\ \}$ given incrementally (on-line) as a sequence of operations $Union(e_1, e'_1), \ldots, Union(e_p, e'_p)$. 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:

- $Union(e, e')$: 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.
- $Find(e)$: returns the current representative of the class containing e.

3 Congruence Closure

3.1 Initial transformations

Although well-known congruence closure algorithms exist, the classical ones of Downey, Sethi and Tarjan [14], Nelson and Oppen [15], and Shostak [16] are not very convenient for our purposes. They are formulated on graphs, and, in order to obtain the best known worst-case complexity bound, $O(n \log n)$, rather involved manipulations are needed; for example, a transformation to graphs of outdegree 2 is applied, see [14].

Since our DPLL procedure will call the congruence closure module a large number of times, and since we will extend our procedure to richer logics, we prefer to replace this transformation by another cleaner one, at the formula representation level, which is done *once and for all*, already on the input formula given to our $DPLL(EUF)$ procedure.

Our initial transformation consists of *Curryfying* the terms, and then, as in [17,18], introducing new constant symbols c for giving names to non-constant subterms t. These two steps will produce an equivalent problem whose size is linear with respect to the original one.

3.1.1 Transformation into Curry Terms

The first step of our initial transformation consists of Curryfying all input terms. After that, there will be only one binary "apply" function symbol (denoted in the following by f) and constants. More formally: consider a new signature $\mathcal{F}'$ obtained from the original $\mathcal{F}$ by introducing a new binary function symbol "f", and converting all other symbols into constants. Then the *Curry form* of a term t in $T(\mathcal{F})$ is a term $Curry(t)$ in $T(\mathcal{F}')$ defined as follows:

$$\begin{cases} Curry(c) = c, \quad \text{if } c \text{ is a constant symbol, and} \\ Curry(g(t_1 ... t_n)) = f(\dots f(\,f(g, Curry(t_1)), Curry(t_2)), \dots, Curry(t_n)) \end{cases}$$

For example, the Curry form of $g(a, h(b), c))$ is $f(f(f(g, a), f(h, b)), c)$. Similarly, we consider the *Curry* transformation on equations, where $Curry(s = t)$ is $Curry(s) = Curry(t)$, and also on sets of equations, where $Curry(E)$ is $\{Curry(eq) \mid eq \in E\}$. We make the following simple observations:

Proposition 3 *Let t be a term. Then $|Curry(t)| \leq 2|t| - 1$, i.e., the Curry transformations only produces a linear growth of the input.*

Proposition 4 *Let E be a set of equations over $\mathcal{F}$ and let $s = t$ be an equation over $\mathcal{F}$. Then Curry(E) $\models$ Curry($s = t$) if, and only if, $E \models s = t$.*

3.1.2 Flattening into Terms of Depth at Most 2

The second step consists of introducing new constant symbols c for giving names to non-constant subterms t; such t are then replaced everywhere by c and the equation $t = c$ is added. More formally, consider the following transformation step on E:

$$E \Rightarrow E' \cup \{c = t\} \qquad\qquad (\textit{Constant introduction and replacement})$$

where c is a fresh constant symbol and E' is obtained by replacing all occurrences of t in E by c.

For example, we flatten the equation $f(f(f(g, a), f(h, b)), b) = b$ by replacing it by $\{\ f(g, a) = c,\ f(h, b) = d,\ f(c, d) = e,\ f(e, b) = b\ \}$.We have the following:

Proposition 5 *Let E_0 be a set of equations, let $s = t$ be an equation, (both built over $\mathcal{F}'$), and let E be obtained by applying zero or more constant introduction and replacement steps on E_0. Then the following holds:*

(1) $E_0 \models s = t$ if, and only if, $E \models s = t$.
(2) If a and b are constants not occurring in $E \cup \{s = t\}$, then $E \models s = t$ if, and only if, $E \cup \{s = a, t = b\} \models a = b$.
(3) By applying a linear number of constant introduction and replacement steps to E_0 an E can be obtained such that all equations of E have a constant side, E has depth at most 2, and $|E| \leq 2|E_0|$.

As a consequence of Proposition 5 we can assume that our congruence closure module receives as input only equations between two constants or between a constant and a "f" applied to two constants. In a *DPLL*(*EUF*) setting, this transformation is done once and for all on the initial problem. After this, the atoms in our EUF formula will be (dis)equalities between constants: all equations involving function symbols will have already be sent to the congruence closure module. However, these transformations can also be done back and forth at each call to our the congruence closure procedure.

3.2 Incremental congruence closure

Here we will define an *incremental* congruence closure algorithm: we are given a sequence of equations over $T(\mathcal{F}')$ intermixed with questions about whether two terms s, t over $T(\mathcal{F}')$ are currently congruent. Formally, the abstract data type we will consider for incremental congruence closure stores a set of equations E_0 and supports the following operations:

- *Merge*$(t{=}c)$: the equation $t{=}c$ is added to E_0. We require t to be either a constant or a flat term of the form $f(a, b)$.
- *AreCongruent?*(s, t) : returns "*yes*" if s and t currently belong to the same congruence class, i.e., $E_0 \models s{=}t$, and "*no*" otherwise.

3.3 Implementation

The procedure uses the following five simple data structures, which induce the equivalence class representation. As said, each equivalence class is identified by its *representative*, which is a certain constant of the class. The procedure we will present maintains the invariants for data structures 2, 3, 4 and 5 described below:

(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 in $\{1, 2\}$. In both cases, we inserting such an element in *Pending*, if the merge of the constants a and b is pending. The need of adding pairs of the form $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$ instead of simply adding $a{=}b$ will be clear in Section 5. When needed, *Pending* will be seen as a set of equations. In this case, a pair $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$ will represent the equation $a{=}b$.
(2) The *Representative* table: an array indexed by constants, containing for each constant its current representative.
(3) The *Class lists*: for each representative, the list of all constants in its class.
(4) The *Use lists*: for each representative a, *UseList*(a) is the 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).
(5) The *Lookup table*: for all pairs of representatives (b, c), *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 whenever such an equation exists. Otherwise, *Lookup*(b, c) is $\perp$.

Since initially no input equations have been processed the data structures *UseList* and *Pending* are initialized as empty and *Lookup*(a, b) is $\perp$ for all pairs (a, b). For each constant a, *ClassList*(a) is initialized to contain only a and *Representative*(a) is set to a. Note that *Lookup* could be stored in a hash table, since a 2-dimensional array will be almost empty. 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. **For each** c in *ClassList*(*old_repr_a*) **Do**
18. set *Representative*(c) to b'
19. move c from *ClassList*(*old_repr_a*) to *ClassList*(b')
20. **For each** $f(c_1, c_2)$=c in *UseList*(*old_repr_a*)
21. **If** *Lookup*(c'_1, c'_2) is some $f(d_1, d_2)$=d **Then**
22. add ($f(c_1, c_2)$=c, $f(d_1, d_2)$=d) to *Pending*
23. remove $f(c_1, c_2)$=c from *UseList*(*old_repr_a*)
24. **Else**
25. set *Lookup*(c'_1, c'_2) to $f(c_1, c_2)$=c
26. move $f(c_1, c_2)$=c from *UseList*(*old_repr_a*) to *UseList*(b')

Informally, each iteration of *Propagate()* picks a pending equation. If the equation is not redundant, lines 18 and 19 add it to the union-find data structure with eager path compression. Lines 20-26 traverse the *UseList* of the constant whose representative has changed and, checking the *Lookup* table, detect new pairs of constants to be merged.

The *AreCongruent*?(s, t) function is much simpler. It only checks whether the function *Normalize* described below rewrites both s and t into the same term.

```
Procedure Normalize(t)
  If t is a constant Then
    return t'
  Else /* t is f(t1, t2) */
    u1 :=Normalize(t1)
    u2 :=Normalize(t2)
    If u1 and u2 are constants and Lookup(u1, u2) is f(a1, a2) = a Then
      return a'
    Else
      return f(u1, u2)
```

In order to be used in an SMT setting, a congruence closure procedure needs to be backtrackable. For that purpose in our implementation we used a mixed policy which proved to be efficient in practice. The changes on the *Representative* and *ClassList* data structures are stacked in order to be undone, but the *Lookup* table is not restored under backtrack, but instead has time stamps to indicate whether its information is valid or not.

3.4 Correctness

In this section we prove the correctness of the previously presented algorithm. Similarly to Section 3.5, these results are presented for the self-containedness of the paper and can be skipped by the expert reader.

The aim of the algorithm is to process a set of equations E_0 and to be able to answer whether an equation belongs to the congruence closure of E_0. Internally, the procedure *Merge* will compute the congruence closure of the input equations in the following standard form:

Definition 6 *A set of equations E is in* standard *form if its equations are of the form $a = b$ or of the form $f(a, b) = c$ whose (respective) left hand side terms a and $f(a, b)$ only occur once in E.*

Definition 7 *Let E_0 be a set of equations of the form $a = b$ or of the form $f(a, b) = c$. A* standard congruence closure *for E_0 is a set of equations E in standard form such that $E_0 \equiv E$.*

In the following, again $a', b', c', \dots$ denote the current representatives of the constants $a, b, c, \dots$. The set of equations already input to our algorithm is denoted by E_0. At any point of the algorithm, we denote by *RepresentativeE* the set of all non-trivial equations of the form $a = a'$ and of the form $f(a', b') = c'$ where a, b and c are constants in E_0 and *Lookup*(a', b') is $f(c_1, c_2) = c$ for some constants c_1 and c_2.

We will prove that after the set E_0 of input equations has been processed, *RepresentativeE* is a standard congruence closure for E_0.

Lemma 8 *Apart from the invariants of the data structures 2, 3, 4, and 5, the following invariants hold each time line 13. is executed and after each call to Merge.*

Inv1: RepresentativeE is in standard form

Inv2: $(\mathit{RepresentativeE} \cup \mathit{Pending})^* = E_0^*$

PROOF. Invariant *Inv1* always holds by definition of *RepresentativeE*. The invariants of the data structures 2, 3, 4, and 5, as well as invariant *Inv2* trivially hold before any input equation is processed. Given an input equation $s{=}t$, lines 3 and 7 make sure that the equation is added to *Pending*. If lines 10 and 11 are executed the equation is added to *RepresentativeE* and also *Lookup* and *UseList* are modified to preserve the data structure invariants. Hence if *Propagate()* preserves all invariants, so it does the procedure *Merge*.

To see that *Propagate()* preserves all invariants, we check lines 14, 18, 19, 22, 23, 25 and 26, which are the only ones that modify the data structures, and show that the congruence $(\mathit{RepresentativeE} \cup \mathit{Pending})^*$ is changed by no iteration: (i) each time an equation is removed from *Pending* (line 14.), lines 18 and 19. ensure that this equality will belong to the next *RepresentativeE*, and also preserve the invariants of the data structures 2 and 3; (ii) all equations $c = d$ that are added to *Pending* (at line 22.) are in the previous $(\mathit{RepresentativeE} \cup \mathit{Pending})^*$: if the input equation $f(c_1, c_2) = c$ is processed it is because, say, c_1 (the reasoning is the same for c_2) has changed its representative from a' to b', and $f(a', c_2')$ and $f(b', c_2')$ were congruent to c and d respectively in the previous $(\mathit{RepresentativeE} \cup \mathit{Pending})^*$. Moreover, we can remove $f(c_1, c_2) = c$ from the *UseList* (at line 23.) because *old_repr_a* is no longer a representative; (iii) lines 25 and 26 ensure that $\mathit{Lookup}(a', b')$ is defined for all input terms $f(a, b)$ and that the *UseList* for each representative contains all needed equations, i.e., they preserve the invariants for *UseList* and *Lookup*. □

Now the following result easily follows:

Theorem 9 *After a set of equations E_0 has been processed, the following holds:*

(1) RepresentativeE is a standard congruence closure for the input E_0.
(2) For any two terms s and t, AreCongruent?(s, t) returns "yes" if and only if $s{=}t$ is in the congruence closure of E_0.

PROOF. For the first claim, note that when the procedure terminates *Pending* is always empty. Then, since $(\mathit{RepresentativeE} \cup \mathit{Pending})^* = E_0^*$ by invariant *Inv2*, we know that also $\mathit{RepresentativeE}^* = E_0^*$. Finally, since by invariant *Inv1* *RepresentativeE* is in standard form, *RepresentativeE* is a standard congruence closure for the input E_0.

For the second claim, note that *RepresentativeE*, with the equations oriented from left to right, is a convergent term rewrite system for E_0 and $\mathit{AreCongruent?}(s, t)$ simply checks whether s and t have the same normal form. □

3.5 Runtime analysis

Theorem 10 *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 $\mathit{AreCongruent?}(s, t)$ can be answered in $O(|s|+|t|)$ and the space required for the whole sequence is $O(n)$.*

PROOF. 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 18 and 19 is executed in total $O(m \log m)$ times, namely when some constant changes its representative. For each one of the m constants this cannot happen more than $\log m$ times, because the size of its class is at least doubled each time and is upper bounded by m. In the loop at lines 21–26, 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.

Each question $\mathit{AreCongruent?}(s, t)$ amounts to computing the normal form of s and t, and to checking whether they are syntactically equal. *Normalize*(t) clearly takes linear time in t and comparing the two canonized terms can also be done in time $O(|s| + |t|)$, since the canonization of a term never has size larger than the term itself. □

4 Producing Explanations from UF

In this section we extend the classical union-find 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 $(e_1,e'_1)\dots(e_p,e'_p)$ 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 11 *Given the numbered sequence of Union operations:*

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

a call to Explain(e_1,e_4) *returns the explanation* $\{(e_7,e_1),(e_{10},e_7),(e_{10},e_4)\}$. $\square$

Since we assume that no redundant union is processed, we have the following:

Proposition 12 *The subset E returned by Explain is unique if it exists.*

PROOF. Consider the undirected graph which has as edges the pairs in the sequence U of unions. Since U includes no redundant unions, this graph has no cycles. It is easy to see that *Explain*(e,e') consists exactly of the edges on the unique path between e and e'. $\square$

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 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 $e \rightarrow e'$.

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

Example 14 *(Example 11 revisited).*

Assume again that 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):

$$\begin{array}{ccccccccc}
 & & 8 \rightarrow 1 & \rightarrow & 7 \leftarrow 2 & \qquad & 12 \rightarrow & 9 & \rightarrow 3 \rightarrow 13 \\
 & & & \nearrow & \uparrow & & & \uparrow & \\
14 \rightarrow 11 \rightarrow 4 \rightarrow 10 & & & & 6 & & & 5 &
\end{array}$$

□

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 15 *In a sequence of $m \geq n$ finds and $n-1$ intermixed unions, the previous data structure performs 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 [19].

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

5 Producing Explanations from Congruence Closure

In this section we use the union-find data structure presented in Section 4 in order to extend the congruence closure algorithm of Section 3 to support the following operation:

- *Explain*(c_1, c_2): assume a sequence M of merges $(s_1, t_1) \dots (s_p, t_p)$ has occurred, and that (c_1, c_2) is in the congruence closure of M; then *Explain*(c_1, c_2) returns a subset $E = \{(s_{i_1}, t_{i_1}) \dots (s_{i_k}, t_{i_k})\}$ of M, with $1 \leq i_1 < \dots < i_k \leq p$, such that exactly at the i_k-th merge c_1 and c_2 became congruent, due to the merge operations in E.

The only addition to the congruence closure algorithm presented in Section 3.3 is to consider the proof forest mentioned in the previous section, which stores the necessary information to implement the required *Explain* operation. The procedures *Merge*, *AreCongruent*? and *Normalize* do not need to be changed, whereas the *Propagate* procedure is slightly modified by adding the necessary information to the proof forest (the framed line 6 in the following algorithm):

1. **Procedure** *Propagate()*
2.   **While** *Pending* is non-empty **Do**
3.   Remove E of the form $a{=}b$ or $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$ from *Pending*
4.   **If** $a' \neq b'$ and, wlog., $|ClassList(a')| \leq |ClassList(b')|$ **Then**
5. $old\_repr\_a := a'$
6. $\boxed{\text{Insert edge } a \to b \text{ labelled with } E \text{ into the } \textit{proof forest}}$
7.   **For each** c in *ClassList*(old_repr_a) **Do**
8.   set *Representative*(c) to b'
9.   move c from *ClassList*(old_repr_a) to *ClassList*(b')
10.   **For each** $f(c_1, c_2){=}c$ in *UseList*(old_repr_a)
11.   **If** *Lookup*(c'_1, c'_2) is some $f(d_1, d_2){=}d$ **Then**
12.   add $(f(c_1, c_2){=}c, f(d_1, d_2){=}d)$ to *Pending*
13.   remove $f(c_1, c_2){=}c$ from *UseList*(old_repr_a)
14.   **Else {**
15.   set *Lookup*(c'_1, c'_2) to $f(c_1, c_2){=}c$
16.   move $f(c_1, c_2){=}c$ from *UseList*(old_repr_a) to *UseList*(b')

Although in *Explain*(c_1, c_2) we assume c_1 and c_2 to be constants, we will see that the previous operation can be extended to *Explain*(s, t) for arbitrary terms s and t with only extra $O(\,|s| + |t|\,)$ time. Another important thing to remark is that sometimes multiple explanations are possible. Our choice here is to return the oldest possible explanation since returning the shortest explanation is NP-hard and even returning *irredundant* explanations is not easy, as we will see in Section 5.4.

5.1 Complexity due to proof forest maintenance

Since only *Propagate()* is modified, by adding line 6, the complexity of the procedure *AreCongruent*? will not be changed. As for *Merge*, note that during a sequence of n *Merge* operations, line 6. will be executed every time two classes are merged, that is, at most m times, where m is the number of constants. Each time, the only work to be done is the reorientation of the appropriate edges in the *proof forest*. Since there are at most m edges and each edge is reoriented at most $O(\log m)$ times, line 6. will only add $O(m \log m)$ to the total complexity, which will remain $O(n \log n)$. Hence the complexity results in Theorem 10 also apply here.

5.2 Implementation of Explain

The *Explain* operation will be performed on the proof forest previously defined. As said, each edge $a - b$ is labelled with a single input equation $a{=}b$ or with a pair of input *structural* equations $(f(a_1, a_2){=}a, f(b_1, b_2){=}b)$. Intuitively, the information on the label represents the reasons why the edge was added. 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 4, by an array of pointers. The way *Explain* is implemented is described informally by means of the following example:

Example 16 *Below we show a numbered sequence of six 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:*

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

$$a \xrightarrow{1,3} d \xleftarrow{2} c \xleftarrow{4} e \xleftarrow{5} b \xleftarrow{6} h$$

On an Explain(a,b) operation, the nearest common ancestor d is detected, and the merge operations on the paths $a \rightsquigarrow d$ (1,3) and $b \rightsquigarrow d$ (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 $b \rightsquigarrow d$ 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 $b \rightsquigarrow d$ 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.* □

Although the algorithm seems quite simple, avoiding such repeated visits is a little bit tricky. The solution we propose uses an additional union-find data structure in the following way.

The Additional Union-Find, and *HighestNode*. At each call to *Explain*, an additional union-find data structure is reset to keep track of the constants that are already equivalent by the proof output so far.If b and d are in such a situation, any subsequent call to *Explain*(b,d) does not have to be processed, since the proof already contains an explanation for this fact. But the situation is more complicated, since also parts of previous subproofs can be reused, as it happened in Example 16. There, *Explain*(h,d) can be seen as the union $h{=}b \cup$ *Explain*(b,d), but since *Explain*(b,d) is already part of the output it does not have to be processed. To detect such situations, in this additional Union-Find, 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, that is, the one which is closest to the root; 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*$(a, parent(a))$ will 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). Note that this is done using the additional Union-Find.

Finding the Nearest Common Ancestor in the Proof Forest. As shown in the example, the first thing to do upon a call to *Explain*(a,b) is to compute the nearest common ancestor of a and b. We consider 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 edges already part of the proof.

Now we can present the two procedures implementing *Explain*:

```
1.  Procedure Explain(c1, c2)
2.    Set PendingProofs to {c1=c2}
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)   /* note that c is already HighestNode(c)
10.   While a≠c Do
11.     b := parent(a)   /* in the additional Union-Find */
12.     If edge a → b is labelled with a single input merge a=b
13.       Output a=b
14.     Else   /* edge labelled with f(a1, a2)=a and f(b1, b2)=b */
15.       Output f(a1, a2)=a and f(b1, b2)=b
16.       Add a1=b1 and a2=b2 to PendingProofs
17.     Union(a, b)   /* in the additional Union-Find */
18.     a := HighestNode(b)
```

5.3 Complexity of Explain

Theorem 17 *For an Explain*(c_1, c_2) *operation, a* k*-step proof can be found in time* $O(k\,\alpha(k,k))$.

PROOF. 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. □

For operations of the form *Explain*(s,t) with arbitrary s and t, one first has to Curryfy and flatten the terms s and t until they have been reduced to constants c_s and c_t, respectively. For each replacement of a term $f(a_1,a_2)$ with a constant a a call to *Merge*$(f(a_1,a_2),a)$ is required. Finally a call *Explain*(c_s,c_t) gives the desired explanation.

Proposition 18 *For an Explain(s,t) with s and t arbitrary terms, an additional $O(|s|+|t|)$ time is required to output a k-step proof.*

PROOF. It is easy to see that the number of constant introduction and replacements is linear in the size of the term. Hence we only have to prove that each call *Merge*$(f(a_1,a_2),a)$ takes constant time. If in the algorithm of section 3.3 *Lookup*(a',b') is $\perp$, lines 10. and 11. are executed in constant time. Otherwise, $(f(a_1,a_2){=}a, f(b_1,b_2){=}b)$ is added to *Pending* for some b_1, b_2 and b and *Propagate* is called. The key point now is to note that the constant a is fresh. This implies that $|ClassList(a')| = 1$ and hence only one iteration of the first loop will be needed. As for the second loop, since a is fresh, its *UseList* will be empty and no iteration will be required. □

5.4 Quality of explanations and experimental results

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*), from now on we will focus on length.

Example 19 *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. □

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 [4] . 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 20 *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:

$$a \rightarrow b \leftarrow c \quad \textit{and} \quad b_1 \rightarrow a_1 \leftarrow c_1$$

Now Explain(a=c) *will return all five equations. However, the two equations containing* b_1 *are redundant.* □

We have run our algorithm as given in the previous subsection over a very large set of benchmarks (all the EUF examples mentioned in [7], available at the second author's home page), producing about 20000 different proofs. There, on average, explanations have 14.9 steps; redundant explanations are returned in 13.92 percent of the cases, having, on average, 51 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 in $\{1,2\}$.

Theorem 21 *A call to Explain*(a,b) *returns a redundant proof only if the proof contains three equations of the same structural class.*

PROOF. Any proof P of a=b can be seen as a set of subproofs of the form

$$x - x_1 - x_2 - \ldots - x_n - y$$

[4] Ashish Tiwari. Personal communication.

where for some subproof x is a and y is b. Moreover, each step $c - d$ is due to (i) an input equation $c{=}d$ or (ii) input equations $f(c_1, c_2){=}c$ and $f(d_1, d_2){=}d$. In the latter case, P must also contain subproofs for $c_1{=}d_1$ and $c_2{=}d_2$.

Let's take a redundant proof P of $a{=}b$ and let P' be an irredundant proof such that $P' \subsetneq P$. There is at least one subproof $x{=}y$ where P and P' differ. If all the steps in this subproof of P' appeared in the proof forest, they would also belong to the corresponding subproof of $x{=}y$ in P. Hence, there must be a step $c{=}d$ in P' not present in the proof forest. Since all steps of type (i) belong to the proof forest, the step has to be of type (ii), involving input equations $f(c_1, c_2){=}c$ and $f(d_1, d_2){=}d$, with c's and d's equivalent in P'. Since $P' \subsetneq P$, we know that $f(c_1, c_2){=}c$ (the same argument holds for $f(d_1, d_2){=}d$) also appears in P and, analyzing the *Explain* procedure it can be seen that the proof forest must include an edge $c - e$ present in P labelled with input equations $f(c_1, c_2){=}c$ and $f(e_1, e_2){=}e$, where e_i is equivalent to c_i for i in $\{1, 2\}$. Hence, $f(c_1, c_2){=}c$, $f(d_1, d_2){=}d$ and $f(e_1, e_2){=}e$ are equations in P that belong to the same structural class. □

The presence of such equations of the same structural class can be checked in time linear in k, and is an extremely good filter: three such equations occur only in 0.13 percent of the irredundant explanations.

The 14 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".

5.5 Proof Forests with Structural Classes as Nodes.

We have also implemented a variant of our proof forest 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 22 *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 (one of them being a single node):

$$[a,b,c] \quad and \quad b_1 \rightarrow a_1 \leftarrow c_1$$

and Explain(a=c) *will return only the structural equations involving* a *and* c *and the corresponding recursive explanation that* a_1=c_1. $\square$

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 3.3 percent of the explanations are still redundant, having on average 37 steps, of which 7 are redundant. Using the test, now postprocessing is needed only in 3.83 percent of the cases. By the stronger test given by Theorem 24 below, it might be possible to remove part of the 0.53 percent of false positives, but this is unlikely to be of any practical relevance.

Example 23 *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:

$$\begin{array}{c} [a,b] \rightarrow a_1 \leftarrow [c,d] \leftarrow d_1 \leftarrow c_1 \\ \uparrow \\ b_1 \end{array}$$

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.* $\square$

Theorem 24 *With the structural classes implementation, a call to Explain*(a,b) *returns a redundant proof only if it contains two equivalent non-singleton structural classes.*

PROOF. Again, take a redundant proof P of a=b and let P' be an irredundant proof such that $P' \subsetneq P$. We can identify a subproof x=y where P and P' differ. Moreover this subproof in P' is such that it includes a step c=d of type (ii) not present in the proof forest nor in the corresponding subproof of P. Let $f(c_1, c_2)$=c and $f(d_1, d_2)$=d, with c's and d's equivalent in P', be the input equations involved in this step. Clearly, c and d do not belong to the same structural class, since the step is not in P.

But, since $P' \subsetneq P$, we know that $f(c_1, c_2)=c$ (the same argument holds for $f(d_1, d_2)=d$) appears in P. If we analyze the *Explain* procedure it must be the case that P includes a step $c - e$ where c and e belong to the same structural class and another step $d - e'$ with d and e' belonging the same structural class. Hence, c and d belong to two different non-singleton equivalent structural classes. $\square$

5.6 Explain on non-Transformed Inputs

Here we show how one can adapt the *Explain* procedure explained above in order to deal with sequences of non-Curryfied, non-flattened equations. The idea is very simple. Each time the congruence closure procedure receives an equation $s=t$, it internally applies Curryfication and flattening. In this process, the terms s ant t will be replaced by constants c_s and c_t, giving rise to the equation $c_s=c_t$. Due to flattening, several other equations of the form $f(a, b)=c$ will also be generated. Now, the only modification in *Explain* is that only equations between constants will be output, but replacing newly introduced constants by the original terms they abstract.

6 Extension to Integer Offsets

Although the logic of EUF is already very useful for the verification of pipelined microprocessors [1], several extensions have been shown to be relevant in practice. In a paper by Bryant, Lahiri, and Seshia [12], the functions *successor (s)* and *predecessor (p)* appear, and all terms are interpreted as integers. We will show that, surprisingly, all previously presented results can be smoothly extended to support these two interpreted function symbols. Moreover, the same time and space bounds can still be obtained and very efficient decision procedures are obtained in practice [7].

6.1 Congruence Closure with integer offsets

In this section we deal with (conjunctions of positive, as before) input equations built over free symbols and *successor* and *predecessor*. To denote a (sub)term t with k successor symbols $s(\dots s(t)\dots)$, we write $t + k$ and similarly write $t + k$ with negative k for $p(\dots p(t)\dots)$. This is why we speak of terms with *integer offsets*. Given a set of equations E over terms with integer offsets, the *congruence closure with integer offsets* of E is the smallest congruence relation $'='$ containing E and such that:

(1) $\forall x \;\; p(s(x)) = x$
(2) $\forall x \;\; s(p(x)) = x$
(3) $\forall x \;\; \underbrace{p(p(...p(}_{n} x)...) \neq x$ for all integers $n > 0$

Note that axioms like $s(s(x)) \neq x$ are not needed: reasoning ad absurdum, if $s(s(x))$=x then $p(p(s(s(x))))$=$p(p(x))$ which, by (1) implies that x=$p(p(x))$, contradicting (3). The first difference with the standard congruence closure problem is that conjunctions of positive equations with integer offsets can be unsatisfiable, that is, the congruence closure with integer offsets does not always exist.

Example 25 *The set* $\{ \; f(a) = c, \; f(b) = c + 1, \; a = b \; \}$ *is unsatisfiable.* $\square$

However, in spite of this difference, we will show that one can still obtain the same time and space bounds as for the case with only uninterpreted symbols. The main idea is to extend the notion of equivalence relation for dealing with *equivalences up to offsets*:

Example 26 *Consider the three equations:*

$$\begin{aligned} a + 2 &= b - 3 \\ b - 5 &= c + 7 \\ c &= d - 4 \end{aligned} \qquad \textit{which can equivalently be written as:} \qquad \begin{aligned} a &= b - 5 \\ b &= c + 12 \\ c &= d - 4 \end{aligned}$$

Here all four constants are equivalent up to some offset. If we take b as the representative of this class, we can write the other constants with their corresponding offsets with respect to the representative b in a class list:

$$\{ \; \mathbf{b} = a + 5 = c + 12 = d + 8 \}$$

thus storing an infinite set of congruence classes, namely the ones represented by $\ldots, b - 1, b, b + 1, \ldots$ in finite space. $\square$

6.1.1 The Initial Transformations

The extension to integer offsets does not affect much the process of Curryfication and flattening. Curryfication is only modified by imposing that for any term t and any integer k we have $Curry(t + k) = Curry(t) + k$ and flattening is not affected at all.

Example 27 *The equation $g(a + 1, h(b + 2), b - 2) = b - 1$ in Curryfied form* *becomes:*

$$f(f(f(g, a+1), f(h, b+2)), b-2) = b-1$$

which is flattened into:

$$\begin{aligned} f(g, a+1) &= c \\ f(h, b+2) &= d \\ f(c, d) &= e \\ f(e, b-2) &= b-1 \end{aligned}$$

□

Note that, due to the fact that the first arguments of the "f" symbol do not represent full (sub)terms of the original input, after the transformation they will have no integer offsets.

Moreover, this property is preserved during the congruence closure process, because the congruence closure process can only make them equal to other such first-argument terms. This fact is illustrated by the following example.

Example 28 *Consider the equations:*

$$\begin{array}{rcrcll} g(a,a,a)=c & \textit{Curry} & f(f(f(g,a),a),a)=c & \textit{flat} & f(g,\,a) = g_1 & f(g,\,b)=g'_1 \\ g(b,b,b)=d & \Longrightarrow & f(f(f(g,b),b),b)=d & & f(g_1,a) = g_2 & f(g'_1,b)=g'_2 \\ a=b & & a=b & \Longrightarrow & f(g_2,a) = c & f(g'_2,b)=d \\ & & & & a = b & \end{array}$$

Here the constant g represents a non-existing 0-ary version of g, and g_1 represents a term $g(a)$ with a unary version of g, which of course also does not exist in the input equations; similarly, g_2 is $g(a,a)$ (a non-existing version of g with 2 arguments). During the congruence closure process, when a is merged with b, the unary, binary and ternary versions of g and g' get merged as well, But note that it is impossible that g_i gets merged with g_j or with g'_j, for $i \neq j$. Roughly speaking, there is a distinct sort for each arity. □

Altogether, we can assume that no integer offsets will ever appear in the first argument of a "f" symbol.

6.1.2 The Algorithm for Integer Offsets

In the following, k with possible subscripts will represent concrete integers. Again, our incremental congruence closure algorithm receives a sequence of equations intermixed with questions about whether two terms s and t are currently congruent. The algorithm stores a set of equations E_0 and supports the following operations:

- *Merge*$(t{=}c{+}k_c)$: the equation $t{=}c{+}k_c$ is added to E_0. If E_0 is inconsistent, it returns *unsatisfiable*. Due to the initial transformations we can assume t to be either a constant or a flat term of the form $f(a, b + k_b)$.
- *AreCongruent?*(s, t) : returns "*yes*" if s and t currently belong to the same congruence class, i.e., $E_0 \models s{=}t$, and "*no*" otherwise.

The data structures used in this case are similar to the ones used in Section 3.

(1) *Pending*: a list whose elements are input equations $a{=}b + k_b$, or pairs of input equations $(f(a_1, a_2 + k_{a_2}) = a + k_1, f(b_1, b_2 + k_{b_2}) = b + k_b)$ where a_1 and b_1 are already congruent, as well as $a_2 + k_{a_2}$ and $b_2 + k_{b_2}$. In both cases, we insert such an element if the merge of the constants a and b modulo the corresponding offset is pending.

(2) The *Representative* table: an array indexed by constants, containing for each constant a, the pair (b, k) such that b is its representative with $b{=}a + k$.

(3) The *Class lists*: for each representative, the list of all pairs (constant, offset) in its class, as in Example 26.

(4) The *Use lists*: for each representative a, *UseList*(a) is the list of input equations $f(b_1, b_2 + k_{b_2}){=}b + k_b$ such that a is the representative of b_1 or of b_2 (or of both).

(5) The *Lookup table*: for all pairs of representatives (b, c) and constant k_c, *Lookup*$(b, c + k_c)$ is some input equation $f(a_1, a_2 + k_{a_2}){=}a + k_a$ such that *Representative*$(a_1) = (b, 0)$, *Representative*$(a_2) = (c, k_{a_2} - k_c)$ iff such an equation exists. Otherwise, *Lookup*$(b, c{+}k_c)$ is $\perp$. Since this would require an infinite three-dimensional table, we store it in a finite hash table.

(6) The *Proof forest*: the same structured presented in Section 4. For each input or derived equation $a{=}b + k_b$ it includes an oriented edge $a \to b$ or $b \to a$. Note that no offset appears in the proof forest.

The initialization is adapted as expected from the case without offsets. In the following, for each constant a, as before we denote its representative constant by a', and now we also write $r(a{+}k_a)$ to denote the representative of such a sum, i.e., $r(a{+}k_a)$ is $a' {+} k_a {-} k$ if *Representative*$(a) = (a', k)$. Similarly, the representative of an equation $a{=}b + k$ is $a'{=}b' + k'$ where k' is $k + k_a - k_b$ if *Representative*$(a) = (a', k_a)$ and *Representative*$(b) = (b', k_b)$.

The algorithm is adapted to support integer offsets as follows:

Procedure *Merge*$(s=t)$
  If s and t are of the form a and $b+k_b$, respectively **Then**
    add $a=b+k_b$ to *Pending*
    return *Propagate()*
  Else /* $s=t$ is of the form $f(a_1, a_2+k_{a_2})=a+k_a$ */
    If *Lookup*$(a'_1, r(a_2+k_{a_2}))$ is some $f(b_1, b_2+k_{b_2})=b+k_b$ **Then**
      add ($f(a_1, a_2+k_{a_2})=a+k_a, f(b_1, b_2+k_{b_2})=b+k_b$) to *Pending*
      return *Propagate()*
    Else
      set *Lookup*$(a'_1, r(a_2+k_{a_2}))$ to $f(a_1, a_2+k_{a_2})=a+k_a$
      add $f(a_1, a_2+k_{a_2})=a+k_a$ to *UseList*(a'_1) and to *UseList*(a'_2)

Procedure *Propagate()*
  While *Pending* is non-empty **Do**
    Remove E of the form $a=b+k_b$ or
        $(f(a_1, a_2+k_{a_2})=a+k_a, f(b_1, b_2+k_{b_2})=b+k_b)$ from *Pending*
    Let $a'=b'+k_{b'}$ be the representative of E
    If $a' \neq b'$ and, wlog., $|ClassList(a')| \leq |ClassList(b')|$ **Then**
      old_repr_a := a'
      Insert edge $a \to b$ labelled with E into the *proof forest*
      For each (c, k_c) in *ClassList*(*old_repr_a*) **Do**
        set *Representative*(c) to $(b', k_c - k_{b'})$
        remove (c, k_c) from *ClassList*(*old_repr_a*)
        add $(c, k_c - k_{b'})$ to *ClassList*(b')
      For each $f(c_1, c_2+k_{c_2})=c+k_c$ in *UseList*(*old_repr_a*)
        If *Lookup*$(c'_1, r(c_2+k_{c_2}))$ is some $f(d_1, d_2+k_{d_2})=d+k_d$ **Then**
          add $(f(c_1, c_2+k_{c_2})=c+k_c, f(d_1, d_2+k_{d_2})=d+k_d)$ to *Pending*
          remove $f(c_1, c_2+k_{c_2})=c+k_c$ from *UseList*(*old_repr_a*)
        Else
          set *Lookup*$(c'_1, r(c_2+k_{c_2}))$ to $f(c_1, c_2+k_{c_2})=c+k_c$
          move $f(c_1, c_2+k_{c_2})=c+k_c$ from *UseList*(*old_repr_a*) to *UseList*(b')
    Else If $a'=b'$ and $k_{b'} \neq 0$
      return *unsatisfiable*

Similarly, the *AreCongruent*?(s,t) procedure follows the same structure as the one in Section 3, but here *Normalize* takes the integer offsets into account when normalizing a term. In order to simplify the presentation, in the procedure *Normalize* we sometimes identify the pair (a, k_a) with $a+k_a$.

Procedure *Normalize(t)*
  If t is a constant $a + k_a$ **Then**
    return $r(a + k_a)$
  Else /* t is $f(t_1, t_2 + k_{t_2})$ */
    (u_1, k_1) :=*Normalize*(t_1) /* note that k_1 will be zero */
    (u_2, k_2) :=*Normalize*$(t_2 + k_{t_2})$
    If u_1, u_2 are cnts and *Lookup*$(u_1, u_2{+}k_2)$ is $f(a_1, a_2{+}k_{a_2}){=}a{+}k_a$ **Then**
      return $r(a + k_a)$
    Else
      return $f(u_1, u_2 + k_2)$

The notions of standard form and of standard congruence extend in the expected way to integer offsets and the corresponding correctness and runtime analysis results, analogous to Theorems 10 and 9, follow along the same lines.

Theorem 29 *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 $O(|s|+|t|)$ and the space required for the whole sequence is $O(n)$.*

Theorem 30 *After a set of equations E_0 has been processed, for any two terms s and t, AreCongruent?(s,t) returns "yes" if and only if $s{=}t$ is in the congruence closure with integer offsets of E_0.*

6.2 Proof-producing congruence closure with integer offsets

The extension of the proof-producing mechanism from the case with no integer offsets is even simpler. The key point is to note that if a and b are in the same equivalence class, they are equivalent up to a *unique* integer offset. Hence there is at most one k_b such that $a = b + k_b$ holds.

Hence, a call to *Explain*$(a, b + k_b)$ can always be reduced to *Explain*(a, b). This way, the algorithm presented in Section 5 needs almost no modification. The only thing to be noted is that if an edge labelled with $(f(a_1, a_2 + k_{a_2}) = a{+}k_a, f(b_1, b_2{+}k_{b_2}){=}b{+}k_b)$ is output, the proofs to be added to *PendingProofs* do not consider the integer offsets, that is, one simply adds $a_1{=}b_1$ and $a_2{=}b_2$.

All the points presented in the case with no integer offsets, such as the possible redundancy of the proofs, the sufficient conditions for irredundancy or the possibility of working with structural classes also apply in the presence of integer offsets.

7 Related Work and Conclusion

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. Due to its simplicity and efficiency, the algorithms here presented have been implemented in two state-of-the-art SMT solvers such as BarcelogicTools and MathSAT [20].

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. This was already mentioned in [5], where the possibility of using a trial-and-error method for finding explanations was considered to be impractical. Instead, they proposed extracting explanations from abstract proofs, which, compared to our approach, is asymptotically worse in theory and produces substantially worse explanations in practice. Another example of this phenomenon is SRI's "lemmas-on-demand" approach in the ICS tool: in [3] 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 [21–23]. In particular, in the paper "Justifying Equality" [23], 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 [23] already holds for our basic algorithm of Section 5. Another interesting approach is the one of [24]. There, only the union-find case is considered. Proofs are built using assumptions and the axioms of *reflexivity, symmetry* and *transitivity.* Given such a proof, it is shown that by means of a term rewrite system, a minimal proof can be obtained in time $O(n^{log_2 3})$.

References

[1] J. R. Burch, D. L. Dill, Automatic Verification of Pipelined Microprocessor Control, in: D. L. Dill (Ed.), 6th International Conference on Computer Aided Verification, CAV'94, Vol. 818 of Lecture Notes in Computer Science, Springer, 1994, pp. 68–80.

[2] A. Armando, C. Castellini, E. Giunchiglia, SAT-Based Procedures for Temporal Reasoning, in: S. Biundo, M. Fox (Eds.), 5th European Conference on Planning, ECP'99, Vol. 1809 of Lecture Notes in Computer Science, Springer, 2000, pp. 97–108.

[3] L. de Moura, H. Rueß, Lemmas on Demand for Satisfiability Solvers, in: 5th International Conference on Theory and Applications of Satisfiability Testing, SAT'02, 2002, pp. 244–251.

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, R. Sebastiani, A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions, in: A. Voronkov (Ed.), 18th International Conference on Automated Deduction, CADE'02, Vol. 2392 of Lecture Notes in Conference Science, Springer, 2002, pp. 195–210.

[5] C. Barrett, D. Dill, A. Stump, Checking Satisfiability of First-Order Formulas by Incremental Translation into SAT, in: E. Brinksma, K. G. Larsen (Eds.), 14th International Conference on Computer Aided Verification, CAV'02, Vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 236–249.

[6] C. Flanagan, R. Joshi, X. Ou, J. B. Saxe, Theorem Proving using Lazy Proof Explanation, in: W. A. H. Jr., F. Somenzi (Eds.), 15th International Conference on Computer Aided Verification, CAV'03, Vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 355–367.

[7] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli, DPLL(T): Fast Decision Procedures, in: R. Alur, D. Peled (Eds.), 16th International Conference on Computer Aided Verification, CAV'04, Vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 175–188.

[8] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM, JACM 7 (3) (1960) 201–215.

[9] M. Davis, G. Logemann, D. Loveland, A Machine Program for Theorem-Proving, Communications of the ACM, CACM 5 (7) (1962) 394–397.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: 38th Design Automation Conference, DAC'01, ACM Press, 2001, pp. 530–535.

[11] C. W. Barrett, L. de Moura, A. Stump, SMT-COMP: Satisfiability Modulo Theories Competition, in: K. Etessami, S. Rajamani (Eds.), 17th International Conference on Computer Aided Verification, CAV'05, Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 20–23.

[12] R. E. Bryant, S. K. Lahiri, S. A. Seshia, Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions, in: E. Brinksma, K. G. Larsen (Eds.), 14th International Conference on Computer Aided Verification, CAV'02, Vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 78–92.

[13] T. T. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to algorithms, MIT Press, 1990.

[14] P. J. Downey, R. Sethi, R. E. Tarjan, Variations on the Common Subexpressions Problem, Journal of the ACM, JACM 27 (4) (1980) 758–771.

[15] G. Nelson, D. C. Oppen, Fast Decision Procedures Based on Congruence Closure, Journal of the ACM, JACM 27 (2) (1980) 356–364.

[16] R. E. Shostak, An Algorithm for Reasoning about Equality, Communications of the ACM, CACM 21 (7) (1978) 583–585.

[17] D. Kapur, Shostak's Congruence Closure as Completion, in: H. Comon (Ed.), 8th International Conference on Rewriting Techniques and Applications, RTA'97, Vol. 1232 of Lecture Notes in Computer Science, Springer, 1997, pp. 23–37.

[18] L. Bachmair, A. Tiwari, Abstract Congruence Closure and Specializations, in: D. A. McAllester (Ed.), 17th International Conference on Automated Deduction, CADE'97, Vol. 1831 of Lecture Notes in Computer Science, Springer, 2000, pp. 64–78.

[19] R. E. Tarjan, Efficiency of a Good but not Linear Set Union Algorithm, Journal of the ACM, JACM 22 (2) (1975) 215–225.

[20] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, R. Sebastiani, The MathSAT 3 System., in: R. Nieuwenhuis (Ed.), 20th International Conference on Automated Deduction, CADE'05, Vol. 3632 of Lecture Notes in Computer Science, Springer, 2005, pp. 315–321.

[21] A. Stump, D. L. Dill, Generating Proofs from a Decision Procedure, in: A. Pnueli, P. Traverso (Eds.), Proceedings of the FLoC Workshop on Run-Time Result Verification, 1999.

[22] R. Klapper, A. Stump, Validated Proof-Producing Decision Procedures, in: 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning, PDPAR'04, Cork, Ireland, 2004.

[23] L. de Moura, H. Rueß, N. Shankar, Justifying Equality, in: 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning, PDPAR'04, Cork, Ireland, 2004.

[24] A. Stump, L. Tan, The Algebra of Equality Proofs, in: J. Giesl (Ed.), 16th International Conference on Term Rewriting and Applications, RTA'05, Vol. 3467 of Lecture Notes in Computer Science, Springer, 2005, pp. 469–483.