Congruence Closure with Integer Offsets

Robert Nieuwenhuis[⋆] and Albert Oliveras[⋆⋆]

Technical University of Catalonia
Jordi Girona 1
08034 Barcelona, Spain
{roberto,oliveras}@lsi.upc.es

Abstract. Congruence closure algorithms for deduction in ground equational theories are ubiquitous in many (semi-)decision procedures used for verification and automated deduction. They are also frequently used in practical contexts where some interpreted function symbols are present. In particular, for the verification of pipelined microprocessors, in many cases it suffices to be able to deal with *integer offsets*, that is, instead of only having ground terms t built over free symbols, all (sub)terms can be of the form $t + k$ for arbitrary integer values k.
In this paper we first give a different very simple and clean formulation for the standard congruence closure algorithm which we believe is of interest on itself. It builds on ideas from the abstract algorithms of [Kap97,BT00], but it is easily shown to run in the best known time, $O(n\ log\ n)$, like the classical algorithms [DST80,NO80,Sho84].
After that, we show how this algorithm can be smoothly extended to deal with integer offsets without increasing this asymptotic complexity.

1 Introduction

Many applications of verification and automated deduction benefit from (semi)-decision procedures for particular theories. For example, in circuit verification one can consider abstractions that forget about the meaning of certain interpreted functions, and then decide the satisfiability of formulae in the so-called logic of equality with uninterpreted functions (EUF) [BD94]. EUF formulae are boolean formulae over atoms that are (dis)equalities between terms without variables. For example, the EUF formula:

$$f(a,b) = f(c,d) \quad \vee \quad a \neq c \quad \vee \quad b \neq d$$

is a tautology, and

$$(\ f(f(a)) \neq b \vee f(f(f(b))) \neq b\ ) \quad \wedge \quad f(a) = a \quad \wedge \quad a = b$$

is unsatisfiable.

[⋆] Both authors partially supported by the spanish CICYT project Maverish ref. TIC2001-2476-C03-01.

[⋆⋆] Supported by a FPU grant, ref.AP2002-3533, from the Spanish Secretaría de Estado de Educación y Universidades.

Deciding the satisfiablity of EUF formulae also has applications for proving satisfiability in first-order logic with equality: checking whether a model of cardinality k exists roughly amounts to instantiating in all possible ways with k new constants and deciding the satisfiability of the resulting ground EUF formula.

Due to the arbitrary boolean part of the formula, the EUF satisfiability problem is obviously NP-hard, and its membership in NP is also easily shown. Currently, most implementations dealing with EUF and its extensions are based on translations into propositional SAT. However, we are working on a general procedure for EUF, and several extensions of it, without translating into SAT. Our algorithm is based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60,DLL62], but where the information coming from the current interpretation is eagerly propagated by incremental constraint solvers. The idea is that these constraint solvers can be plugged in into a general $DPLL(X)$ scheme, very much in the flavour of the constraint logic programming scheme $CLP(X)$. In the case of EUF, which can be seen as $DPLL(=)$, the solver roughly amounts to a *congruence closure* procedure (extended for dealing with backtracking and disequalities), which finds the new equalities that follow from the given ones.

In this paper we concentrate on congruence closure for positive equations. Nowadays well-known congruence closure algorithms were already given in the early 1980s by Downey, Sethi, and Tarjan, [DST80] and also by Nelson and Oppen [NO80]; see also Shostak's method for the combination of decision procedures [Sho84]. However, for two main reasons, these early versions of congruence closure are not very convenient for our purposes.

First, 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 [DST80]. 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, and which is done *once and for all*, already on the input formula given to our $DPLL(=)$ procedure. Our key idea for this is to *Curryfy*, like in the implementation of functional languages; as far as we know, this had not been done before for congruence closure; as a result, there will be only one binary "apply" function symbol (denoted here by a dot $\cdot$) and constants. For example, Curryfying $f(a, g(b), b)$ gives $\cdot(\cdot(\cdot(f, a), \cdot(g, b)), b)$. This idea makes the algorithms surprisingly simple and clean and hence easier to extend and to reason about.

Second, like in the more abstract congruence closure approaches, such as the ones of [Kap97,BT00], we introduce 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. As we will see, then, in combination with Curryfication, one can obtain the same efficiency as in more sophisticated directed acyclic graph (DAG) implementations by appropriately indexing the new constants like c, which play the role of the pointers to the (shared) subterms like t in the DAG approaches. For example, we flatten the equation

$$\cdot(\cdot(\cdot(f,a),\cdot(g,b)),b) = b$$

by replacing it by

$$\{\ \ \cdot(f,a)=c,\ \ \cdot(g,b)=d,\ \ \cdot(c,d)=e,\ \ \cdot(e,b)=b\ \ \}$$

As a consequence of this transformation, which is again done once and for all on the initial problem that is input to our DPLL procedure, we can assume that our congruence closure module receives as input only equations between two constants or between a constant and a "$\cdot$" applied to two constants[1].

Congruence closure algorithms are also frequently used in practical contexts where some interpreted function symbols are present. In particular, for the verification of pipelined and/or superscalar microprocessors, in many cases it suffices to be able to deal with *integer offsets*, that is, instead of only having ground terms t built over free symbols in the equations, all (sub)terms can be of the form $t+k$ for arbitrary integer values k. This has been done in the logic handled by Bryant et al. [BLS02], where predecessor and successor symbols occur.

The remainder of this paper is structured as follows. Section 2 introduces the basic notions and notations. After Section 3, where the two initial transformations are formalized, in Section 4 we give our extremely simple formulation of the congruence closure algorithm, and we prove its correctness and its $O(n\ log\ n)$ runtime and linear space requirements. In Section 5, we give evidence for an additional advantage of this clean algorithm, by showing that it can be extended in very a smooth way for dealing with integer offsets, while maintaining the same time and space requirements. Finally, in Section 6 we conclude and outline our plans for future work in this project.

2 Basic notions and notations

Let $\mathcal{F}$ be a (finite) set of function symbols with an arity function $arity\colon \mathcal{F} \to I\!N$. Function symbols f with $arity(f) = n$ are called *n-ary* symbols (when $n = 1$, one says *unary* and when $n = 2$, *binary*). If $arity(f) = 0$, then f is a *constant symbol*. The set of ground terms over $\mathcal{F}$, denoted by $\mathcal{T}(\mathcal{F})$, is the smallest set containing all constant symbols such that $f(t_1,\ldots,t_n)$ is in $\mathcal{T}(\mathcal{F})$ whenever $f \in \mathcal{F}$, $arity(f) = n$, and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F})$.

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 $|f(t_1,\ldots,t_n)| = 1 + |t_1| + \ldots + |t_n|$. The *depth* of a term s is denoted by $depth(s)$ and is defined: $depth(a) = 1$ if a is a constant symbol and $depth(f(t_1,\ldots,t_n)) = 1 + max(depth(t_1),\ldots,depth(t_n))$.

An *equivalence relation* is a reflexive, symmetric, and transitive binary relation. A relation $=$ on $T(\mathcal{F})$ is *monotonic* if $f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)$ whenever

[1] In fact, after this, the atoms in our EUF formula will be (dis)equalities between constants: all function symbols are hidden inside the congruence closure module.

f is an n-ary function symbol in $\mathcal{F}$ and $s_i = t_i$ for all i in $1 \ldots n$. A *congruence relation* is a monotonic equivalence relation.

A *ground equation* is an (unordered) pair of ground terms (s, t), denoted by $s = t$. Given a set of ground equations E built over $\mathcal{F}$, we denoted by E^* the congruence *generated* by E: the smallest congruence relation $=$ over $T(\mathcal{F})$ containing E. We sometimes write $E \models s = t$ to denote the fact that $s = t$ belongs to E^*, and 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 $E' \models E$.

3 Initial Transformations

3.1 Transformation into Curry terms

Consider a new signature $\mathcal{F}'$ obtained from the original $\mathcal{F}$ by introducing a new binary function symbol "$\cdot$", and converting all other symbols into constants. Then the *Curry form* of a term t in $T(\mathcal{F})$ is a term $\mathit{Curry}(t)$ in $T(\mathcal{F}')$ defined as follows:
$\mathit{Curry}(c) = c$, if c is a constant symbol, and
$\mathit{Curry}(f(t_1 ... t_n)) = \cdot(\ldots \cdot(\ \cdot(f, \mathit{Curry}(t_1)), \mathit{Curry}(t_2)), \ldots, \mathit{Curry}(t_n))$
For example, the Curry form of $f(a, g(b), c))$ is $\cdot(\cdot(\cdot(f, a), \cdot(g, b)), c)$. Similarly, we consider the *Curry* transformation on equations, where $\mathit{Curry}(s = t)$ is $\mathit{Curry}(s) = \mathit{Curry}(t)$, and on sets of equations: $\mathit{Curry}(E) = \{\mathit{Curry}(e) \mid e \in E\}$. We make the following simple observations:

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

Proposition 2. *Let E be a set of ground equations over $\mathcal{F}$ and let $s = t$ be an equation over $\mathcal{F}$. Then $\mathit{Curry}(E) \models \mathit{Curry}(s = t)$ if, and only if, $E \models s = t$.*

3.2 The flattening transformation into terms of depth at most 2

Consider the following transformation step on E:

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

where c is a new constant symbol not occurring anywhere in E and E' is obtained by replacing all occurrences of t in E by c. We have the following:

Proposition 3. *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 $E_0 \models s = t$ if, and only if, $E \models s = t$.

Furthermore, if a and b are constants not occurring in E and in $s = t$, then $E \models s = t$ if, and only if, $E \cup \{s = a, t = b\} \models a = b$.

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 $|E| \leq 2|E_0|$.

4 Congruence Closure

In the following, (possibly primed or indexed) lowercase letters $a, b, c, d, \ldots$ denote constant symbols. The procedure receives as input a set of equations E of the form $a = b$ or of the form $\cdot(a, b) = c$, and we assume that no term $\cdot(a, b)$ occurs more than once in the input (i.e., after flattening no different constant names exist for the same term).

The procedure halts when it has computed the congruence generated by the input equations, after which it can output (in linear time) the list of congruence classes. It can also answer in constant time queries asking whether two terms (constants or terms $\cdot(a, b)$) belong to the same class. In fact, the procedure produces a convergent term rewrite system by which any term t rewrites into its unique normal form in time linear in $|t|$ (see Subsection 4.2); hence this allows one to decide in time $O(|s| + |t|)$ whether two arbitrary terms s and t belong to the same class (note that it is irrelevant whether s and t are built over the original signature or the Curryfied one, since the translation is linear too). This is possible without any post-processing because, unlike other congruence closure algorithms, our procedure does not rely on the well-known union-find data structure (by which equalities between constants cannot always be decided in constant time).

The procedure uses the following five simple data structures, which include the equivalence class representation, where (as usual) each class has a single distinguished *representative* constant:

1. *Pending unions*: a list of pairs of constants yet to be merged.
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 *Lookup table*: for each input term $\cdot(a, b)$, $\mathit{Lookup}(\mathit{Representative}(a), \mathit{Representative}(b))$ returns in constant time a constant c such that $\cdot(a, b)$ is equivalent to c, and returns $\perp$ if there is no such c.
5. The *Use lists*: for each representative a, the list of input equations $\cdot(b, c) = d$ such that a is the representative of b or c (or of both).

These data structures are initialized as expected: *Pending* contains the initial equations of the form $a = b$, and for each initial equation $\cdot(a, b) = c$, it belongs to $\mathit{UseList}(a)$ and to $\mathit{UseList}(b)$, and $\mathit{Lookup}(a, b)$ is c ($\mathit{Lookup}(a, b)$ is undefined for all other pairs (a, b)). *Representative* and *ClassList* contain all constants as their own representatives in one-element classes. In the following algorithm, a' denotes $\mathit{Representative}(a)$ for each constant a:

```
1.  While Pending is non-empty Do
2.    Remove an equation $a = b$ from Pending
3.    If $a' \neq b'$ and, wlog., $|ClassList(a')| \leq |ClassList(b')|$ Then
4.      For each $c$ in $ClassList(a')$ Do
5.        set $Representative(c)$ to $b'$ and add $c$ to $ClassList(b')$
6.      EndFor
7.      For each $\cdot(c, d) = e$ in $UseList(a')$ Do
8.        If $Lookup(c', d')$ is some $f$ and $f' \neq e'$ Then
9.          add $e' = f'$ to Pending
10.       EndIf
11.       set $Lookup(c', d')$ to $e'$
12.       add $\cdot(c, d) = e$ to $UseList(b')$
13.     EndFor
14.   EndIf
15. EndWhile
```

Example 1. Consider the following input and the set E_0 obtained after curryfying and flattening it:

$$\left.\begin{array}{r} f(a) = g(b) \\ g(c) = h(f(c), g(a)) \\ b = c \\ f(c) = g(a) \\ h(d, d) = g(b) \\ g(a) = d \end{array}\right\} \Longrightarrow \begin{bmatrix} \cdot(f, a) = e_1 \\ \cdot(g, b) = e_2 \\ \cdot(g, c) = e_3 \\ \cdot(f, c) = e_4 \\ \cdot(h, e_4) = e_5 \\ \cdot(g, a) = e_6 \\ \cdot(e_5, e_6) = e_7 \\ \cdot(h, d) = e_8 \\ \cdot(e_8, d) = e_9 \end{bmatrix} + \begin{bmatrix} e_1 = e_2 \\ e_3 = e_7 \\ b = c \\ e_4 = e_6 \\ e_9 = e_2 \\ e_6 = d \end{bmatrix}$$

The following eight iterations take place:

1. Let the first equation to be removed from *Pending* be $e_1 = e_2$. We set *Representative*(e_1) to e_2 (although since $|e_1| = |e_2| = 1$ we also had the reverse choice); since *UseList*(e_1) is empty, we can go the next iteration.
2. Now the equation $e_3 = e_7$ is picked from *Pending*, and (again we can choose) set *Representative*(e_3) to e_7; *UseList*(e_3) is also empty.
3. We pick $b = c$ from *Pending*, set *Representative*(b) to c, and now need to handle the single equation $\cdot(g, b) = e_2$ in *UseList*(b): since *Lookup*(g', b') = *Lookup*$(g, c) = e_3$, and $e_3' = e_7$ and $e_2' = e_2$, the equation $e_7 = e_2$ is added to *Pending*; furthermore, *Lookup*(g, c) is set to e_2, and $\cdot(g, b) = e_2$ is added to *UseList*(c).
4. $e_7 = e_2$ is handled (here, *Pending* is a stack) and *Representative*(e_2) is set to e_7; *UseList*$(e_2) = \emptyset$.
5. Pick $e_4 = e_6$ and set *Representative*(e_4) to e_6; the only equation in *UseList*(e_4) is $\cdot(h, e_4) = e_5$, which leads to no new equations.
6. Set *Representative*(e_9) to e_7.

7. Set *Representative*(d) to e_6; *UseList*$(d) = \{\cdot(h,d) = e_8, \cdot(e_8,d) = e_9\}$, from which, due to its first equation, $e_5 = e_8$ is added to *Pending*, because *Lookup*(h',d') is *Lookup*$(h,e_6) = e_5$ and e_5 and e_8 are distinct representatives.
8. Set *Representative*(e_5) to e_8; *UseList*$(e_5) = \{\cdot(e_5,e_6) = e_7\}$, and, since *Lookup*$(e_5',e_6')$ is *Lookup*$(e_8,e_6) = e_9$, a new equality $e_7 = e_9$ follows, but it is discarded because $e_7' = e_9' = e_7$.

The final congruence (with representatives written in bold) is:

$\{\,\mathbf{a}\,\}\quad \{\,\mathbf{c} = b\,\}\quad \{\,\mathbf{f}\,\}\quad \{\,\mathbf{g}\,\}\quad \{\,\mathbf{h}\,\}$
$\{\,\mathbf{e_6} = e_4 = d = \cdot(g,a) \ = \cdot(f,c)\,\}$
$\{\,\mathbf{e_7} = e_1 = e_2 = e_3 = e_9 = \cdot(f,a) = \cdot(g,b) = \cdot(g,c) = \cdot(e_8,d) = \cdot(e_5,e_6)\,\}$
$\{\,\mathbf{e_8} = e_5 = \cdot(h,e_4) \ = \cdot(h,d)\ ;\}$ □

4.1 Runtime analysis

Theorem 1. *The algorithm runs in $O(n\ log\ n)$ time and in linear space.*

Proof. The proof is simple and quite standard. The *Lookup* table can be implemented by a hash table, or, if hashing is not considered appropriate, by a two-dimensional array[2]. The time spent for maintaining the *Representative* data structure that gives constant time access to representatives is amortized over the whole algorithm: the loop at lines 4,5,6 is executed $O(k\ log\ k)$ times, where k (which is –usually much– smaller than the input size n) is the number of different constants, namely each time one of the k constants changes its representative (which cannot happen more than $log\ k$ times, because the size of its class is at least doubled each time and is upper bounded by k). The same happens for the loop at lines 7-13: each one of the at most n input equations of the form $\cdot(c,d) = e$ is treated when c or d changes its representative (which, as before, cannot happen more than $log\ k$ times). This in turn implies that $O(n\ log\ k)$ new equations are added to *Pending* at line 9. Altogether, we obtain an $O(n\ log\ n)$ runtime. Using hashing for the *Lookup* table, only linear space is used. Note that the *UseList*(a') in line 7 is no longer needed and its space can be re-used (otherwise, this would require $O(n\ log\ n)$ space). □

4.2 Correctness

The aim of the algorithm is to compute the congruence generated by the input equations, in the following standard form:

Definition 1. *A set of equations E is in* standard *form if its equations are of the form $a = b$ or of the form $\cdot(a,b) = c$ whose (respective) left hand sides a and $\cdot(a,b)$ only occur once in E.*

[2] To avoid the quadratic initialization time of such a two-dimensional *Lookup* table, one can store in each *Lookup*$[a,b]$ an index k to an auxiliary array A, where $A[k]$ contains $\cdot(a,b) = c$, and with a counter *max* indicating that A contains correct (i.e., initialized) information for all $k < max$.

Intuitively, in such a standard E, the constants at the right hand sides and below the "$\cdot$" symbols are all representatives of their respective classes. In fact, considering its equations (oriented from left to right) as rewrite rules, it is a convergent term rewrite system (see [DP01]); by rewriting with it, deciding whether an equation $s = t$ is in the congruence can be done in time $O(|s = t|)$.

Definition 2. *Let E_0 be a set of equations of the form $a = b$ or of the form $\cdot(a, b) = c$. 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 $a, b, c, \ldots$ denote constant symbols, and their primed versions $a', b', c', \ldots$ denote their current representatives. The set of input equations of the algorithm is denoted by E_0. For a given time line 1 of the algorithm is executed, we denote by *RepresentativeE* the set of all non-trivial equations of the form $a = a'$ and of the form $\cdot(a', b') = c'$ where a, b and c are constants in E_0 and c is $Lookup(a', b')$.

We will prove that when our algorithm terminates, *RepresentativeE* is a standard congruence closure for the input E_0.

Lemma 1. *Apart from the invariants of the data structures 2, 3, 4, and 5, the following are invariants of the main loop of our algorithm, i.e., they hold each time line 1. is executed:*

Inv1: RepresentativeE is in standard form

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

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* hold initially, by the assumptions on E_0. To see that they are also preserved by the loop, we check lines 2, 5, 9, 11, and 12, which are the only ones that modify the data structures, and show that the congruence $(RepresentativeE \cup Pending)^*$ is changed by no iteration: (i) each time an equation $a = b$ is removed from *Pending* (line 2.), line 5. ensures that this equality will belong to the next *RepresentativeE*, and also preserves the invariants of the data structures 2 and 3; (ii) all $e' = f'$ that are added to *Pending* (at line 9.) are in the previous $(RepresentativeE \cup Pending)^*$: if $e' \neq f$, this is because, say, c (the reasoning is the same for d) has changed its representative from a' to b', and $Lookup(a', d')$ and $Lookup(b', d')$ were congruent to e' and f' repectively in the previous $(RepresentativeE \cup Pending)^*$. (iii) lines 11 and 12 ensure that $Lookup(a', b')$ is defined for all input terms $\cdot(a, b)$ and that the use lists for each representative contain all needed equations, i.e., they preserve the representation invariants 3 and 4. □

Now the following result follows easily:

Theorem 2. *When the algorithm terminates, RepresentativeE is a standard congruence closure for the input E_0.*

Proof. The algorithm terminates when *Pending* is empty. Then, since by invariant *Inv2* $(RepresentativeE \cup Pending)^* = E_0^*$, we have $RepresentativeE^* = E_0^*$; since by invariant *Inv1* *RepresentativeE* is in standard form, *RepresentativeE* is a standard congruence closure for the input E_0. □

5 Integer offsets

In a recent paper by Bryant, Lahiri, and Seshia [BLS02] the logic of EUF is extended in several ways. In particular, in some of their formulae coming from the verification of pipelined microprocessors, the functions *successor (s)* and *predecessor (p)* appear, and all terms are interpreted as integers.

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*.

A difference with the standard congruence closure problem is that conjunctions of positive equations with integer offsets can be unsatisfiable:

Example 2. The set $\{\ f(a)=c,\ f(b)=c+1,\ a=b\ \}$ is unsatisfiable. □

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 free symbols. The main idea is to extend the notion of equivalence relation for dealing with *equivalences up to offsets*:

Example 3. Consider the three equations:

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

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 $\dots, b-1, b, b+1, \dots$ in finite space. □

5.1 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 4. The equation $f(a+1, g(b+2), b-2) = b-1$ in Curryfied form becomes:

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

which is flattened into:

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

□

Note that, due to the fact that the first arguments of the "·" 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 5. Consider the equations:

$$\begin{array}{r} f(a,a,a)=c\\ f(b,b,b)=d\\ a=b \end{array} \quad \overset{\text{Curry}}{\Longrightarrow} \quad \begin{array}{r} \cdot(\cdot(\cdot(f,a),a),a)=c\\ \cdot(\cdot(\cdot(f,b),b),b)=d\\ a=b \end{array} \quad \overset{\text{flat}}{\Longrightarrow} \quad \begin{array}{rr} \cdot(f,\,a)=f_1 & \cdot(f,\,b)=f'_1\\ \cdot(f_1,a)=f_2 & \cdot(f'_1,b)=f'_2\\ \cdot(f_2,a)=c & \cdot(f'_2,b)=d\\ a=b & \end{array}$$

Here f represents a non-existing 0-ary version of f, and f_1 represents a term $f(a)$ with a unary version of f, which of course also does not exist in the input equations; similarly, f_2 is $f(a,a)$ (a non-existing version of f with 2 arguments). The same happens for f'_1 and f'_2. During the congruence closure process, when a is merged with b, the unary versions of f and f' get merged as well, and also the binary versions, as well as, finally, the 3-ary versions, represented by c and d. But note that it is impossible that f_i gets merged with f_j or with f'_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 "·" symbol.

5.2 The algorithm for integer offsets

In the following, possibly subindexed k will represent concrete integers and $a, b, c, d, \ldots$ will be constants. The input for our procedure will be a set of equations E of the form $a = b + k$ or of the form $\cdot(a, b + k_b) = c + k_c$, and we again assume that no term $\cdot(a, b + k)$ occurs more than once in the input.

The procedure for dealing with integer offsets halts giving the congruence generated by E whenever E is satisfiable. When it is not, it returns *unsatisfiable*.

The data structures used in this case are nearly the same as in the previous section:

1. *Pending unions*: a list of equalities of the form $a = b+k$ yet to be processed.
2. The *Representative* table: an array indexed by constants, containing for each constant a, the pair (b, k) such that b is its representative constant, with $b = a+k$.
3. The *Class lists*: for each representative, the list of all pairs (constant, offset) in its class, as in Example 3.
4. The *Lookup table*: for each input term $\cdot(a, b+k_b)$, where $\mathit{Representative}(a) = (a', 0)$ and $\mathit{Representative}(b) = (b', k_{b'})$ (that is, b is $b' - k_{b'}$), the function $\mathit{Lookup}(a', b'+(k_b-k_{b'}))$ returns in constant time $c+k_c$ such that $\cdot(a, b+k_b)$ is equivalent to $c+k_c$, and returns $\perp$ if there is no such $c+k_c$.
5. The *Use lists*: for each representative a, the list of input equations $\cdot(b, c+k_c) = d+k_d$ such that a is the representative of b or c (or of both).

The initialization is as 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 also we 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 $\mathit{Representative}(a) = (a', k)$. The algorithm is as follows:

```
1.  While Pending is non-empty Do
2.    Remove $a = b+k$ with representative $a' = b'+k_{b'}$ from Pending
3.    If $a' \neq b'$ and, wlog., $|\mathit{ClassList}(a')| \leq |\mathit{ClassList}(b')|$ Then
4.      For each $c+k_c$ in $\mathit{ClassList}(a')$ Do
5.        set $\mathit{Representative}(c)$ to $(b', k_c-k_{b'})$ and add it to $\mathit{ClassList}(b')$
6.      EndFor
7.      For each $\cdot(c, d+k_d) = e+k_e$ in $\mathit{UseList}(a')$ Do
8.        If $\mathit{Lookup}(c', r(d+k_d))$ is $f+k_f$ and $r(f+k_f) \neq r(e+k_e)$ Then
9.          add $e = f+(k_f-k_e)$ to Pending
10.       EndIf
11.       set $\mathit{Lookup}(c', r(d+k_d))$ to $r(e+k_e)$
12.       add $\cdot(c, d+k_d) = e+k_e$ to $\mathit{UseList}(b')$
13.     EndFor
14.   ElseIf $a' = b'$ and $k_{b'} \neq 0$
15.     return unsatifiable
16.   EndIf
17. EndWhile
```

Theorem 3. *The algorithm for congruence closure with integer offsets runs in $O(n \log n)$ time and in linear space.*

Proof. The proof is analogous to the one of Theorem 1. In this case the *Lookup* table has to be implemented by a hash table, since the *Lookup* function now in fact has three arguments (constant, constant, offset), unless one could in advance determine the range of the offsets. □

The notions of standard form and of standard congruence extend in the expected way to integer offsets, and the corresponding result, analogous to Theorem 2, follows along the same lines.

6 Conclusions

We have given a new congruence closure algorithm. Apart for being short, clean and simple, it combines the efficiency of the classical algorithms [DST80,NO80] [Sho84] with the elegance of the more modern abstract view, as expressed in the frameworks of [Kap97,BT00,BTV]. These frameworks for abstract congruence closure have led to new insights and new results for several problems in rewriting, but only provide sub-obtimal (quadratic) time complexity results (although we have recently learned that the algorithm of [BT00] can be extended with less abstract and less simple control data in its inference rules in order to make it run in $O(n\ log\ n)$ time; in particular, this requires an on-the-fly ordering on constants that forces congruence classes to be merged in the right order; this is sketched in the journal version [BTV], to appear).

Possibly this work can be seen as a concrete implementation of some of these abstract approaches, but for which a tighter and simpler complexity analysis has become possible, essentially due to the initial Curryfication.

On the other hand, the existing algorithms for ground Knuth-Bendix completion (which implicitly also compute a congruence closure) are all rather involved, and moreover either quadratic, like [PSK96], or are based on the previous use of one of the classical congruence closure algorithms on graphs [Sny89].

We believe that our cleaner formulation of congruence closure will also be useful for improving its explanation and understanding, and for applications and extensions such as the ones we have mentioned. Regarding practical aspects, from our first experiments our algorithm appears to be fast, in spite of the fact that it is extremely easy to implement. We are currently working on the design and a first implementation of the whole *DPLL*(=) procedure, including, of course, the congruence closure module, as well as on its extension to *EUF* with integer offsets. Further extensions to be studied include the presence of other interpreted symbols, like associative and/or commutative ones.

In the paper by Bryant, Lahiri, and Seshia [BLS02] the logic with integer offsets is also further extended with an ordering predicate $>$. But surprisingly, already when only positive atoms $s > t$ are added to the input E_0, deciding the satisfiability of such E_0, which we call here the *CC-Ineq problem* (for congruence closure with inequalities), becomes NP-complete:

Proposition 4. *The CC-Ineq problem is NP-complete.*

Proof. Here we only sketch NP-hardness. Given a graph $G = (V, E)$ where $V = \{a_1, \ldots, a_n\}$ and $E = \{(b_1, b'_1), \cdots, (b_m, b'_m)\}$ and an integer k, the following CC-Ineq formula F_G is satisfiable if and only if G is k-colorable:

$$G(c+1, c+1) = G(c+2, c+2) = \ldots = G(c+k, c+k) = \mathit{true}$$

$$\begin{array}{lcl}
c+k+1 > f(a_1) > c & \qquad & \mathit{true} > G(f(b_1), f(b'_1)) \\
c+k+1 > f(a_2) > c & & \mathit{true} > G(f(b_2), f(b'_2)) \\
\vdots \qquad \vdots \quad \vdots & & \vdots \qquad\qquad \vdots \\
c+k+1 > f(a_n) > c & & \mathit{true} > G(f(b_m), f(b'_m))
\end{array}$$

Intuitively, f represents the colour of each vertex (k possibilites), and G is used to express that no two adjacent vertices will have the same colour. $\square$

References

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. 6th International Computer Aided Verification Conference*, pages 68–80, 1994.

[BLS02] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Procs. 14th Intl. Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, July 27–31 2002.

[BT00] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David McAllester, editor, *Conference on Automated Deduction, CADE '2000*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78, Pittsburgh, PA, June 2000. Springer-Verlag.

[BTV] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*. To appear.

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

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

[DP01] Nachum Dershowitz and David Plaisted. Rewriting. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers and MIT Press, 2001.

[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.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[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.

[PSK96] David A. Plaisted and Andrea Sattler-Klein. Proof lengths for equational completion. *Information and Computation*, 125(2):154–170, 15 March 1996.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[Sny89] Wayne Snyder. An O(n log n) algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations E. In N. Dershowitz, editor, *Rewriting Techniques and Applications, 3th International Conference*, LNCS. Springer-Verlag, 1989.