Deciding equality in the constructor theory*

Pierre Corbineau

pierre.corbineau@cs.ru.nl
Institute for Computing and Information Science
Radboud University Nijmegen, Postbus 9010
6500GL Nijmegen, The Netherlands

Abstract. We give a decision procedure for the satisfiability of finite sets of ground equations and disequations in the *constructor theory*: the terms used may contain both uninterpreted and constructor function symbols. Constructor function symbols are by definition injective and terms built with distinct constructors are themselves distinct. This corresponds to properties of (co-)inductive type constructors in inductive type theory. We do this in a framework where function symbols can be partially applied and equations between functions are allowed. We describe our algorithm as an extension of congruence-closure and give correctness, completeness and termination arguments. We then proceed to discuss its limits and extension possibilities by describing its implementation in the Coq proof assistant.

Among problems in equational reasoning, a crucial one is the word problem: does a set of equations entail another one? In 1947, Post and Markov [15, 7] showed that this is undecidable. What is decidable is whether an equation between *closed* terms is the consequence of a *finite* conjunction of equations between *closed* terms. This problem is called congruence-closure and its decidability was established by Nelson and Oppen [11], and Downey, Sethi, and Tarjan [6]. The algorithms that are described have a quasi-linear complexity. Nelson and Oppen, and also Shostak, described methods to extend the scope of this closed case with some specific equational theories: lists, tuples ...

In general, semi-decision procedures over equations use syntactic methods (replacement, rewriting ...). Hence the algorithm used for congruence-closure is an exception since its principle is to directly build a set of term equivalence classes. The link between congruence-closure and syntactic methods was established by the concept of abstract congruence closure [2, 1].

Because of its simplicity and efficiency, this algorithm makes a good candidate for an implementation of a Coq tactic. Moreover, we wish to extend the decision procedure in order to cover specific properties of constructors for inductive and co-inductive types in Coq:

- Injectivity : $C\,x_1\,\ldots\,x_n = C\,y_1\,\ldots\,y_n \Rightarrow x_1 = y_1 \wedge \cdots \wedge x_n = y_n$

* This work was done at project ProVal, Pole Commun de Recherche en Informatique du plateau de Saclay: CNRS, École Polytechnique, INRIA, Universit Paris-Sud

– Discrimination : $C\,x_1\,\ldots\,x_n \neq D\,y_1\,\ldots\,y_p$

The Coq proof assistant already contained, on the one hand, the `injection` and `discriminate` tactics, that deal with immediate consequences of an equation between constructor-based terms, and on the other hand the `autorewrite` tactic that allows us to normalize a term with respect to a set of oriented equations. Those tactics cannot cooperate with each other, which we could do by designing a tactic for congruence-closure with the constructor theory.

In 2001, I [4] proved in Coq the correctness of the congruence-closure algorithm and designed a tagging system for the data structure allowing us to extract proofs of the equations. That method was implemented in the Coq system in 2002 as a tactic named `congruence`. The proof production method used there is similar to the one discovered and studied in detail by Nieuwenhuis and Oliveras [12].

In 2003, I implemented an extended version of the `congruence` tactic that included the constructor theory. No theoretical ground was given to support that extension, so this is the purpose of this article. Our aim is thus to describe a decision procedure that can decide the combination of the theories of constructors and uninterpreted symbols, by extending the congruence-closure algorithm. This is done by proving that the decision problem can be reduced to a finite set of terms. Moreover, the higher-order logic of the Coq system advocates for solving the problem for simply typed terms, allowing equations in functional types.

In this article, we will only manipulate closed terms (without variables), that we will abusively call *terms* for clarity purposes. We will also talk about *term algebra* instead of *closed term algebra*.

1 The congruence closure problem

The language for the theory of equality uses a distinguished binary relation symbol $\approx$, which is supposed to be well-sorted (i.e. $\approx$-related terms have the same sort). Hence this polymorphic relation symbol is a notation for a set of monomorphic predicates $(\approx_s)_{s\in\mathcal{S}}$ where $\mathcal{S}$ is the set of sorts.

In order to define the congruence-closure problem, we will first explain what a congruence is.

Definition 1 (Congruence). *A well-sorted relation $\approx$ over a first-order term algebra is called a congruence if, and only if, the following conditions hold :*

1. *$\approx$ is an equivalence relation, i.e. satisfies these three rules :*

$$\frac{}{t \approx t}\ \text{REFL} \qquad \frac{s \approx t}{t \approx s}\ \text{SYM} \qquad \frac{s \approx u \qquad u \approx t}{s \approx t}\ \text{TRANS}$$

2. *Any n-ary function symbol f in the signature is a $\approx$-morphism, i.e. it satisfies the following congruence rule :*

$$\frac{s_1 \approx t_1 \qquad \ldots \qquad s_n \approx t_n}{f(s_1,\ldots,s_n) \approx f(t_1,\ldots,t_n)}\ \text{CONGR}_f$$

An interesting property is that congruence relations are stable under arbitrary intersection, thus for any well-sorted binary relation R over a term algebra, there is a unique minimal congruence $\approx_R$ such that $R \subseteq \approx_R$: $\approx_R$ is the intersection of all congruences coarser than R (there is at least one, which relates all terms of the same sort).

The congruence-closure problem can be stated in two equivalent ways:

- Is the finite set of propositions $\{s_1 \approx t_1, \ldots, s_n \approx t_n, u \not\approx v\}$ satisfiable in the theory of equality ?
- Is there a congruence that satisfies both $s_1 \approx t_1, \ldots, s_n \approx t_n$ and $u \not\approx v$?

The problem can be generalized to any number of inequalities $u \not\approx v$, the issue being to determine if one among them contradicts the equations $s \approx t$.

The equivalence between the two questions relies on two arguments: first, if the set of propositions is satisfiable, by an interpretation I, then the $\approx$ relation defined by $s \approx t \Leftrightarrow I(s) = I(t)$ is a congruence such that $s_i \approx t_i$ for any i and $u \not\approx v$. Second, if we have a congruence $\approx$ such that $s_i \approx t_i$ for any i and $u \not\approx v$, then we can build an interpretation mapping any term to its $\approx$ equivalence class. The interpretation is well-defined since $\approx$ satisfies the congruence rule, and it satisfies the equations and the inequality.

The key fact in the proof of congruence-closure decidability is : if T is the set of terms and subterms appearing in the problem instance, then any proof derivation of the equation $u \approx v$ from the axioms using the rules REFL, SYM, TRANS and CONGR, can be turned into a proof where only terms in T are used. From the dual point of view, any interpretation satisfying locally the set of axioms can be extended to the set of all terms.

Nelson and Oppen [10, 11] and Downey, Sethi, Tarjan [6], start from this property to build algorithms representing equivalence classes by a forest of trees (UNION-FIND structure), in order to obtain an optimal complexity.

1.1 Simply typed term algebra

Here, we wish to deal with simply typed terms, so we will use an encoding corresponding to curryfied terms. This representation allows to represent partial function application. This is done by restricting our first-order signatures to a small subset of simply typed signatures.

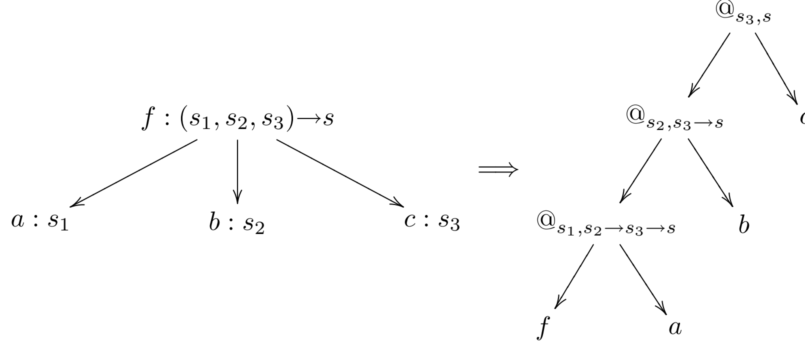
Definition 2 (Simply typed term). *A simply typed term algebra is defined by giving a signature $(\mathcal{S}, \Sigma)$ such that:*

- $\mathcal{S} = \bigcup_{n\in\mathbb{N}} \mathcal{S}_n$*, where* $\mathcal{S}_{n+1} = S_n \cup \{s \rightarrow s' | (s, s') \in \mathcal{S}_n \times \mathcal{S}_n\}$ *and* $\mathcal{S}_0$ *is a set of base sorts* $\alpha_1, \ldots, \alpha_n$*. The sorts in* $\mathcal{S} \setminus \mathcal{S}_0$ *are called* functional sorts.
- Σ *contains only constants and a set of binary function symbols* $@_{s,s'}$ *called* application symbols *which take one argument* $s \rightarrow s'$*, the second in* s *and yields a result in* s'*.*

In practice, only a finite number of the sorts and application symbols will be used since only a finite number of sorts will occur in a given problem. As an abuse of notation, we will often omit the indices of application symbols, and we will shorten $@(@(f,x),y)$ as $(f\,x\,y)$. Any first-order signature can be turned into a simply typed signature by applying a curryfication mapping as follows:

Definition 3 (Curryfied signature). *Let $(\mathcal{S},\Sigma)$ be a first-order signature. The corresponding currified signature is $(\hat{\mathcal{S}},\hat{\Sigma})$, where $\hat{\mathcal{S}}$ is the set of simple sorts generated by using $\mathcal{S}$ as the set of base sorts, and $\hat{\Sigma}$ contains, for each function symbol $f:(s_1,\dots,s_n)\rightarrow s\in\Sigma$, a constant symbol $\hat{f}:s_1\rightarrow(\dots\rightarrow(s_n\rightarrow s))$, along with the n application symbols required to apply $\hat{f}$: $@_{s_1,s_2\rightarrow\dots\rightarrow s},\dots,@_{s_n,s}$.*

From now on, we suppose the symbol $\rightarrow$ is right-associative and will omit unnecessary brackets. Every term t with sort s in the first-order term algebra Σ can be mapped to a curryfied term $\hat{t}$ with the same sort in the term algebra generated by $\hat{\Sigma}$, by following the next example :



The reverse statement, however, is only true for terms with a base sort — note that many simply typed signatures are not curryfied signature, e.g. if they have symbols with sort $(\mathbb{N}\rightarrow\mathbb{N})\rightarrow\mathbb{N}$ — and this gives us a bijection between first-order terms and curryfied terms.

Example 1. The first-order signature for Peano arithmetic has $\mathcal{S}_0=\{\mathbb{N}\}$ as sorts and the following symbols:

$$\begin{array}{l}\{\mathbf{0}:()\rightarrow\mathbb{N}\,;\\ \ \mathrm{S}:(\mathbb{N})\rightarrow\mathbb{N}\,;\\ \ +:(\mathbb{N},\mathbb{N})\rightarrow\mathbb{N}\,;\\ \ \times:(\mathbb{N},\mathbb{N})\rightarrow\mathbb{N}\}\end{array}$$

The sorts used in the corresponding curryfied signature are

$$\mathcal{S}=\{\mathbb{N}\,;\mathbb{N}\rightarrow\mathbb{N}\,;\mathbb{N}\rightarrow\mathbb{N}\rightarrow\mathbb{N}\}$$

and the set of curryfied symbols is :

$$\begin{array}{rl}\{ \quad \hat{\mathbf{0}} & : \mathbb{N}\,; \\ \hat{\mathrm{S}} & : \mathbb{N} \to \mathbb{N}\,; \\ \hat{+} & : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\,; \\ \hat{\times} & : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\,; \\ @_{\mathbb{N},\mathbb{N}} & : (\mathbb{N} \to \mathbb{N}, \mathbb{N}) \to \mathbb{N}\,; \\ @_{\mathbb{N},\mathbb{N}\to\mathbb{N}} & : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}, \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \}\end{array}$$

Please note that we can express functional equations in the curryfied signature, such as: $@(\hat{+}, @(\hat{\mathrm{S}}, \hat{\mathbf{0}})) \approx \hat{\mathrm{S}}$.

We might think that allowing to express more intermediate properties (higher-order equations) in the curryfied framework would allow us to prove more properties in the theory of equality, but this is not the case:

Lemma 1 (Conservativity of curryfication). *The curryfication is conservative with respect to congruence-closure, i.e. any derivation of $\hat{u} \approx \hat{v}$ from $\hat{s}_1 \approx \hat{t}_1, \ldots, \hat{s}_n \approx \hat{t}_n$ in the curryfied system can be turned into a derivation in the first-order system. The converse also holds.*

Proof. By explicit tree transformation, only termination is tricky. Please note that by construction, curryfied terms are fully applied.

2 Extension to constructor theory

We now wish to extend the historical algorithm in order to deal with datatype constructors. Those constructors are injective function symbols, similarly to tuples constructors. They also have a discrimination property: two different constructors in the same sort produce distinct objects. These properties are built into the system Coq's type theory: they hold for constructors of both inductive and co-inductive type.

We do not claim here that those two properties are sufficient to characterize those constructors. For example, inductive type constructors also have acyclicity properties in the style of Peano's axiom about the successor function S, which satisfies $\forall x, x \not\approx S\,x$. These acyclicity properties have been studied in [14].

Other kinds of deductions can be based on the assumption that datatypes are totally generated by their constructors: considering a finite type such as the booleans, with two constant constructors T and F, we can deduce that the set of propositions $\{g\,T \approx a, g\,F \approx a, g\,c \not\approx a\}$ in unsatisfiable. This kind of property is not included in our notion of constructor theory, which would be closer to Objective Caml [13] sum types when taking in account the possibility of raising exceptions.

Definition 4 (Signature with constructors). *A signature with constructors is a simply typed signature with some distinguished constants called constructors. A constructor with shape $C_i^\alpha : \tau_1 \to (\ldots \to (\tau_n \to \alpha))$ where alpha is a base sort is called a n-ary constructor* of the sort α.

By convention, we will now use lowercase letters for regular function symbols and uppercase letters for constructor symbols. The notation C_i^α will refer to some constructor of the base sort α, with the convention that identical constructors will always have identical indices. We now have to explain a congruence satisfying the constructor theory.

Definition 5 (Constructor theory). *A congruence $\approx$ satisfies the constructor theory over the signature Σ if, and only if, the following conditions hold:*

- *$\approx$ satisfies all instances of the following injectivity rule :*

$$\frac{C_i^\alpha\, s_1 \ldots s_n \approx C_i^\alpha\, t_1 \ldots t_n}{s_i \approx t_i}\ \textsc{inj}_i \quad \textit{if} \begin{cases} C_i^\alpha\, s_1 \ldots s_n : \alpha \\ C_i^\alpha\, t_1 \ldots t_n : \alpha \end{cases}$$

- *For any pair of terms $C\, s_1 \ldots s_n$ and $D\, t_1 \ldots t_p$ in sort α, such that C and D are distinct constructors, $C\, s_1 \ldots s_n \not\approx D\, t_1 \ldots t_p$.*

Since we allow equations between terms in any functional sort, we need to give a semantic notion of constructor-generated term relatively to a congruence $\approx$. We will call such terms *inductive term.*

Definition 6 (Inductive term). *A term t is* inductive *with respect to $\approx$ if, and only if, $t \approx C$ for some constructor C or $t \approx (f x)$ for some term x and inductive term f.*

We can immediately remark that if s is inductive w.r.t. $\approx$ and $s \approx t$, then t is also inductive. This allows us to use the notion of *inductive equivalence class*, which contains (only) inductive terms.

The problem we now wish to solve is the satisfiability of a finite set E of ground equations and inequalities of terms inside a constructor theory, which is equivalent to finding a congruence satisfying both the constructor theory and E.

Similarly to basic congruence-closure, we can see that congruences satisfying the *injectivity* of constructors are closed under arbitrary intersection. We can conclude that there exists a smallest congruence satisfying the equations of a given problem and the injectivity rule. If there is a congruence satisfying this problem in the constructor theory, then this smallest congruence also does since it satisfies less inequalities.

In order to use those facts to design our algorithm, we must check that we are able to reduce the decision problem to a finite set of terms, as is done for the congruence-closure problems. Unfortunately, the set of terms and subterms from the problem is not sufficient to decide our problem, as shown by the following example.

Example 2. Let α, β be two base sorts, f,C two symbols in sort $\beta\to\alpha$, D in sort α, b in sort β, among which C and D are (distinct) constructors of the sort α. Then $\{C \approx f; f\, b \approx D\}$ is unsatisfiable, since it allows to prove that $C\, b \approx D$, but the proof uses the term $C\, b$ which does not appear in the initial problem.

$$\dfrac{\dfrac{C \approx f \qquad \dfrac{}{b \approx b}\,\textsc{refl}}{C\, b \approx f\, b}\,\textsc{congr}_@ \qquad f\, b \approx D}{C\, b \approx D}\,\textsc{trans}$$

In order to cope with this, we define application-closed sets of terms. As we deal with closed terms, a sort can be *empty* in a signature Σ. This is the case if no closed term can be built from symbols in Σ, e.g. if $\Sigma = \{f : \beta \to \beta\}$ then β is an empty sort. The following definition takes this remark into account.

Definition 7 (Application-closed set). *A set T of simply typed terms with constructors is application-closed with respect to the congruence $\approx$ if for each inductive equivalence class $\mathcal{C}$ in $T/_{\approx}$ with a functional sort $\tau \to \tau'$, if the sort τ is non-empty then there exist terms $t \in \mathcal{C}$ and $u \in T$ such that $(tu) \in T$.*

Now, we can define a local version of the constructor theory which uses a secondary relation $\rightsquigarrow$, in order to keep the term set considered as small as possible. The $\rightsquigarrow$ is just an asymmetric and external version of $\approx$ that allows its left-hand side to be a term with a constructor in the head position that is not mentioned in the problem. It is actually this *local theory* that will be decided by the algorithm.

Definition 8 (Local constructor theory). *Let T be a subterm-closed set of terms and $\approx$ a well-sorted relation on T. $\approx$ is said to be a* local congruence *on T if it is an equivalence relation on T satisfying every instance of the* CONGR@ *rule where terms occuring are in T.*

Furthermore, a local congruence $\approx$ satisfies locally the constructor theory if, and only if, there exists a relation $\rightsquigarrow$ satisfying the following rules :

$$\frac{C_i^\alpha \approx f}{C_i^\alpha \rightsquigarrow f}\ \textsc{promotion} \qquad \frac{C_i^\alpha\, t_1 \ldots t_k \rightsquigarrow u \quad u\, t_{k+1} \approx v}{C_i^\alpha\, t_1 \ldots t_k\, t_{k+1} \rightsquigarrow v}\ \textsc{compose}$$

$$\frac{C_i^\alpha\, s_1 \ldots s_n \rightsquigarrow u \quad C_i^\alpha\, t_1 \ldots t_n \rightsquigarrow u}{s_i \approx t_i}\ \textsc{injection}$$

and such that there is no term $t \in T$ such that $C_i^\alpha\, s_1 \ldots s_n \rightsquigarrow t$ and $C_j^\alpha\, t_1 \ldots t_p \rightsquigarrow t$, where $i \neq j$ and n and p are the respective arities of C_i^α and C_j^α.

The theorem for completeness of the original congruence-closure algorithm states that any local congruence is the restriction of a congruence on all terms. Now we have this similar local notion for constructor theory, that we use to formulate the theorem that restricts our decision problem to a finite set of terms.

Theorem 1 (Finite set restriction). *Let E be a finite set of equations and inequalities and T be a subterm-closed set of terms containing all the terms occuring in E.*

Let $\approx$ be a local congruence on T satisfying E and satisfying locally the theory of constructors, such that T is application-closed for $\approx$.

Then there exists a congruence $\approx^*$ over the whole term algebra, which, on the one hand, satisfies E and the theory of constructors, and on the other hand, is a conservative extension of $\approx$, i.e. for all terms u, v in T, $u \approx^* v \Leftrightarrow u \approx v$.

Proof. In order to prove this, we explicitly construct a binary tree model for all terms and define the relation $\approx^*$ as relating terms having the same interpretation in the model. We build this model so that $\approx$ is the restriction of $\approx^*$ to T.

Let S be the set of binary trees with leaves labelled by the set $T/_\approx \cup \{e_\tau\}_{\tau in \mathcal{S}}$ where $T/_\approx$ is the set of equivalence classes modulo $\approx$ and and e_τ is an arbitrary object in sort τ, which will be the representation for unconstrained terms.

We also restrict S to contain only *well-sorted trees*, where every leaf and node has a additional sort label, and for any node with sort τ, there is a sort τ' such that the left son of the tree has sort $\tau' \to \tau$ and the right son has sort τ'. We also require that e_τ is labelled by the sort τ and that any equivalence class is labelled by the sort of the terms it contains. Let S_τ be the subset of trees whose root has sort τ.

The notation (s, t) stands for the well-sorted tree with s and t as (well-sorted) subtrees. A tree is said to be *inductive* if it is an equivalence class containing an inductive term (for $\approx$), or if it is not just a leaf.

We can then define the $App_{\tau,\tau'}$ function from $S_{\tau\to\tau'} \times S_\tau$ to $S_{\tau'}$ as follows: suppose we have $a \in S_{\tau\to\tau'}$ and $b \in S_\tau$.

- if $a = e_{\tau\to\tau'}$, then $App_{\tau,\tau'}(e_{\tau\to\tau'}, b) = e_{\tau'}$
- if a is not a leaf then $App_{\tau,\tau'}(a, b) = (a, b)$
- if $a \in T/_\approx$ then :
  - if $b \in T/_\approx$ and there is $t_a \in a$ and $t_b \in b$ such that $(t_a\, t_b) \in T$, then let c be the equivalence class of $(t_a\, t_b)$, we put $App_{\tau,\tau'}(a, b) = c$. This definition is not ambiguous since $\approx$ is a congruence.
  - otherwise
    - if a is inductive, then $App_{\tau,\tau'}(a, b) = (a, b)$
    - else $App_{\tau,\tau'}(a, b) = e_{\tau'}$.

Our interpretation domain is the set S. The interpretation of constants and constructors is their equivalence class in $T/_\approx$: $I(f) = \bar{f}$, and we set $I(@_{\tau,\tau'}) = App_{\tau,\tau'}$. Then $\approx^*$ is defined by $s \approx^* t \Leftrightarrow I(s) = I(t)$.

Now first, $\approx^*$ is a congruence relation since it is by construction an equivalence relation, it is well-sorted since the S_τ sets are disjoint, and since $I(@(f, x)) = App(I(f), I(x))$, $\approx$ satisfies the congruence rule.

Again by construction, terms in T are interpreted into their equivalence class for $\approx$, so $\approx^*$ is conservative with respect to $\approx$, and satisfies E.

Then we show that $\approx^*$ satisfies the constructor theory, which is a bit technical, and too long to be detailed here, see [5], Theorem 2.13, pages 42–45 for the full proof. This concludes the proof. □

Now that we have reduced our problem to a more manageable one, we will give a description of the algorithm itself.

3 The decision algorithm

3.1 Description

We now give an algorithm to decide problems with ground equations and inequalities and a constructor theory. This algorithm is very similar to the one in

[6], except for the part concerning constructors. This algorithm is given by a set of transition rules (read from top to bottom) between quintuples (T, E, M, U, S) made of :

T: Set of **T**erms not up to date in the signature table and for marking
E: Set of unprocessed **E**quations ($\approx$-constraints)
M: Set of unprocessed **M**arks ($\rightsquigarrow$-constraints)
U: **U**NION-FIND structure (map from terms to equivalence classes)
S: **S**ignature table

An equivalence class is a quadruple (r, L, R, C) made of:

r: the **r**epresentative term for the class
L: Set of **L**eft fathers of the class
R: Set of **R**ight fathers of the class
C: Set of **C**onstructor marks of the equivalence class, i.e. set of terms $\rightsquigarrow$-related to terms in the class.

We write $U(u) = (\bar{u}, L(u), R(u), C(u))$ to simplify notation. A *left* (resp. *right*) *father* of a class c is a term whose left (resp. right) subterm is in the class c.

The UNION-FIND structure is best described as a forest of trees in which all nodes contain a term and point to their father node, and the roots contain an additional class quadruple. The class searching operation is done by following the path from a node to the root. The union(U, s, t) operation merges two equivalence classes by making $\bar{s}$ an immediate child of $\bar{t}$, thus making $\bar{t}$ the representative term for all terms in the former class of s, the sets of left and right fathers are then merged, but $\bar{t}$ keeps its original set C.

The *signature table* acts as a cache. It maps a pair of representative terms $(\bar{u}, \bar{v})$ to a term whose left (resp. right) immediate subterms are in the class of u (resp. v), if such a term is registered, and otherwise it yields $\perp$. The suppr(S, R) operation removes bindings to terms in R from the table.

We start with a set $\mathcal{A}$ of equations and disequalities. The set $\mathcal{A}\!\downarrow$ is the set of terms and subterms occuring in $\mathcal{A}$. The set $\mathcal{A}^+$ (resp. $\mathcal{A}^-$) is the set of pairs of terms appearing in an equation (resp. inequality) in $\mathcal{A}$.

The transition rules for the algorithm are given in Figure 1. The INIT rule builds the initial structure, where all terms are apart, all equations and all constructor marks are pending in E and M.

Three issues are possible for the algorithm: the REFUTE rule detects if an inequality in $\mathcal{A}$ has been proved false, it answers that the problem is unsatisfiable. The CONFLICT rule concludes the computation if two terms made of distinct fully applied constructors are proven equal. If there is no constraint left to propagate and none of the above rules apply, then the problem is declared satisfiable using the OK rule.

The MERGE and NO-MERGE rules are used to deal with pending equality constraints, the NO-MERGE rule avoids unnecessary operations. The MERGE rule has to expunge all outdated information from the tables: remove fathers of merged terms from the signature table and put them in T to re-process them later.

The MATCH and MARK rules allow to mark an equivalence class with the inductive terms that belong to it. The MATCH rule propagates the consequences of the injectivity rule to E. The $\text{UPDATE}_{1/2}$ rules are used to update the signature table, propagate the consequences of the congruence rule to E, and propagate marks from subterms to their left fathers.

Example 3. This first example is a very simple one used to illustrate how the signature table works. In the U set, the notation $s \mapsto t$ stands for $\bar{s} = t$, and by default $\bar{u} = u$. Similarly, $C(t) = \emptyset$ for all representative terms not mentioned.

Consider the signature $\mathcal{S}_0 = \{s\}, \Sigma = \{a : s, f : s \to s\}$ and the following problem : $\{f\,a \approx a, f(f\,a) \not\approx a\}$. A possible execution of the algorithm is given in Figure 2.

Example 4. In order to illustrate the use of marks for constructors on equivalence classes, we now give an example with both injection and discrimination. Consider the following signature :

$$\mathcal{S}_0 = \{\texttt{bool}, \texttt{box}\},$$
$$\Sigma = \{g : \texttt{bool} \to \texttt{box}, B : \texttt{bool} \to \texttt{box}, T : \texttt{bool}, F : \texttt{bool}\}$$

The satisfiability problem $\{B \approx g, g\,T \approx g\,F\}$ is solved by our procedure in Figure 3.

3.2 Properties

Theorem 2 (Termination). *Any successive application of the rules of the algorithm reaches SAT or NOSAT in a finite number of steps.*

Proof. By checking the rules, we remark that there is a rule that we can apply in every situation and we can check that the tuples decrease for the lexicographic ordering made of:

1. the number of equivalence classes in the UNION-FIND structure.
2. the union $T \cup M$, ordered by the multiset extension of the size of term sorts, considering that identical sorts are bigger in T than in M.
3. the size of E

Theorem 3 (Correctness). *The algorithm is correct, i.e. if it answers NOSAT then the problem is unsatisfiable.*

Proof. The proof is made by showing that the rules preserve key invariants and that the side-conditions for final transitions ensure correctness. The detailed proof can be found in [5], section 2.3.4, p. 51.

Completeness is proven in two steps: first, we prove that if the algorithm reaches SAT then the equivalence relation induced by the UNION-FIND structures, together with the marking relation, gives a congruence that satisfies locally the constructor theory.

initialization rule

$$\frac{Sat(\mathcal{A})?}{T_0, E, M, U, \emptyset}\ \textsc{Init}\begin{cases} T_0 = \{@(u,v) \in \mathcal{A}\!\downarrow\} \\ E = \mathcal{A}^+ \\ M = \{(C_i^\alpha, C_i^\alpha) | C_i^\alpha \in \mathcal{A}\!\downarrow\} \\ U = u \mapsto (u, \{@(u,v) \in A\!\downarrow\}, \{@(v,u) \in A\!\downarrow\}, \emptyset) \end{cases}$$

conclusion rules

$$\frac{\emptyset, \emptyset, \emptyset, U, S}{SAT}\ \textsc{Ok}\left\{\forall (u,v) \in \mathcal{A}^-, \bar{u} \neq \bar{v}\right.$$

$$\frac{T, E, M, U, S}{NOSAT}\ \textsc{Refute}\left\{\exists (u,v) \in \mathcal{A}^-, \bar{u} = \bar{v}\right.$$

$$\frac{T, E, M \cup \{(t, C_i^\alpha\, \boldsymbol{u})\}, U, S}{NOSAT}\ \textsc{Conflict}\begin{cases} C_i^\alpha \boldsymbol{u} : \alpha \text{ (fully applied)} \\ C_j^\alpha \boldsymbol{v} \in C(t) \\ i \neq j \end{cases}$$

deduction rules

$$\frac{T, E \cup \{(s,t)\}, M, U, S}{\begin{array}{c} T \cup L(s) \cup R(s), E, M \cup \{(t,c) | c \in C(s)\}, \\ \text{union}(U, s, t), \text{suppr}(S, L(s) \cup R(s)) \end{array}}\ \textsc{Merge}\left\{\bar{s} \neq \bar{t}\right.$$

$$\frac{T, E \cup \{(s,t)\}, M, U, S}{T, E, M, U, S}\ \text{no-}\textsc{Merge}\left\{\bar{s} = \bar{t}\right.$$

$$\frac{T, E, M \cup \{(t, C_i^\alpha\, \boldsymbol{u})\}, U, S}{T, E \cup \{(\boldsymbol{u}, \boldsymbol{v})\}, M, U, S}\ \textsc{Match}\begin{cases} C_i^\alpha \boldsymbol{u} : \alpha \text{ (fully applied)} \\ C_i^\alpha \boldsymbol{v} \in C(t) \end{cases}$$

$$\frac{T, E, M \cup \{(t, C_i^\alpha\, \boldsymbol{u})\}, U, S}{T \cup L(t), E, M, U\{C(t) \leftarrow C(t) \cup C_i^\alpha\, \boldsymbol{u}\}, S}\ \textsc{Mark}\begin{cases} C_i^\alpha \boldsymbol{u} : \tau \rightarrow \tau' \text{ (partially applied)} \\ \quad \text{or} \\ C(t) = \emptyset \end{cases}$$

$$\frac{T \cup \{t\}, E, M, U, S}{T, E \cup \{(t,s)\}, M \cup \{(t, F\, v) | F \in \mathcal{C}\}, U, S}\ \textsc{Update}_1\begin{cases} t = @(u,v) \\ S(\bar{u}, \bar{v}) = s \\ C(u) = \mathcal{C} \end{cases}$$

$$\frac{T \cup \{t\}, E, M, U, S}{T, E, M \cup \{(t, F\, v) | F \in \mathcal{C}\}, U, S \cup \{(\bar{u}, \bar{v}) \mapsto t\}}\ \textsc{Update}_2\begin{cases} t = @(u,v) \\ S(\bar{u}, \bar{v}) = \bot \\ C(u) = \mathcal{C} \end{cases}$$

Fig. 1. Rules of our algorithm for the problem $\mathcal{A}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{SAT(f\,a \approx a, f(f\,a) \not\approx a)?}{\{f\,a, f(f\,a)\}, \{(f\,a, a)\}, \emptyset, \emptyset, \emptyset}\ \textsc{Init}}{\{f\,a, f(f\,a)\}, \emptyset, \emptyset, \{f\,a \mapsto a\}, \emptyset}\ \textsc{Merge}}{\{f(f\,a)\}, \emptyset, \emptyset, \{f\,a \mapsto a\}, \{(f, a) \mapsto f\,a\}}\ \textsc{Update}_2}{\emptyset, \{(f(f\,a), f\,a)\}, \emptyset, \{f\,a \mapsto a\}, \{(f, a) \mapsto f\,a\}}\ \textsc{Update}_1 \quad S(\bar{f}, \overline{f\,a}) = f\,a}{\emptyset, \emptyset, \emptyset, \left\{ \begin{array}{ll} f\,a & \mapsto a \\ f(f\,a) & \mapsto a \end{array} \right\}, \{(f, a) \mapsto f\,a\}}\ \textsc{Merge}}{NOSAT}\ \textsc{Refute} \quad \overline{f(f\,a)} = a = \bar{a}$$

Fig. 2. Satisfiability computation for $\{f\,a \approx a, f(f\,a) \not\approx a\}$

Lemma 2 (Local completeness). *If the algorithm reaches SAT, then the relation $\approx$ defined by $u \approx v \Leftrightarrow \bar{u} = \bar{v}$ is a local congruence in $\mathcal{A}\!\downarrow$ which satisfies locally the constructor theory.*

To establish global completeness, we need a hypothesis: the set $\mathcal{A}\!\downarrow$ has to be application-closed for $\approx$. This is not true in general. However the algorithm can be adapted to detect if this occurs, and we can try to add terms to our term set in order to have the terms we want fully applied. The incompleteness detection is done in the following way :

- Initially, every class has *UNKNOWN* status.
- When a class with *UNKNOWN* status is marked with a partially-applied constructor, it goes to *INCOMPLETE* status.
- When an Update rule is applied to a term $(f\,x)$ and the class of f has *INCOMPLETE* status, it goes to *COMPLETE* status.
- When classes are merged, they get the most advanced status of the two merged classes, considering *COMPLETE* most advanced, *INCOMPLETE* less advanced and *UNKNOWN* least advanced.

That way, when the algorithm reaches SAT, the final set is application-closed if, and only if, no class is left with *INCOMPLETE* status. This said, global completeness can be stated as follows:

Theorem 4 (Completeness). *Let $\mathcal{A}$ a non-empty finite set of axioms, if the algorithm reaches SAT from $\mathcal{A}$ and the final set is application-closed for $\approx$, then $\mathcal{A}$ is satisfiable.*

Proof. By using the previous lemma together with Theorem 1.

If all sorts are inhabited, we can add to the final state of the algorithm the partially applied inductive terms, applied to arbitrary terms in the correct sort. Once this is done, a re-run of the algorithm from this state will yield an application-closed set of terms. Thus there is no need to carry on this process more than once. Nevertheless, the first run is necessary to determine how to complete the set of terms.

$$SAT(B \approx g, g\,T \approx g\,F)?$$

INIT

$$\{g\,T, g\,F\}, \{(B,g),(g\,T, g\,F)\}, \{(T,T),(F,F),(B,B)\}, \emptyset, \emptyset$$

MERGE

$$\{g\,T, g\,F\}, \{(g\,T, g\,F)\}, \{(T,T),(F,F),(B,B)\}, \{B \mapsto g\}, \emptyset$$

MERGE

$$\{g\,T, g\,F\}, \emptyset, \{(T,T),(F,F),(B,B)\}, \left\{\begin{array}{c} g\,T \mapsto g\,F \\ B \mapsto g \end{array}\right\}, \emptyset$$

MARK

$$\{g\,T, g\,F\}, \emptyset, \{(F,F),(B,B)\}, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & \end{array}\right\}, \emptyset$$

MARK

$$\{g\,T, g\,F\}, \emptyset, \{(B,B)\}, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \end{array}\right\}, \emptyset$$

MARK

$$\{g\,T, g\,F\}, \emptyset, \emptyset, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \\ & C(g)=\{B\} \end{array}\right\}, \emptyset$$

UPDATE$_2$

$$\{g\,F\}, \emptyset, \{(g\,T, B\,T)\}, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \\ & C(g)=\{B\} \end{array}\right\}, \emptyset$$

MARK

$$\{g\,F\}, \emptyset, \emptyset, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \\ C(g\,F)=\{B\,T\} & C(g)=\{B\} \end{array}\right\}, \emptyset$$

UPDATE$_2$

$$\emptyset, \emptyset, \{(g\,F, B\,F)\}, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \\ C(g\,F)=\{B\,T\} & C(g)=\{B\} \end{array}\right\}, \emptyset$$

MATCH $\quad B\,F \approx B\,T$

$$\emptyset, \{(F,T)\}, \emptyset, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(F)=\{F\} \\ C(g\,F)=\{B\,T\} & C(g)=\{B\} \end{array}\right\}, \emptyset$$

MERGE

$$\{g\,F\}, \emptyset, \{(F,F)\}, \left\{\begin{array}{cc} g\,T \mapsto g\,F & C(T)=\{T\} \\ B \mapsto g & C(g)=\{B\} \\ F \mapsto T & C(g\,F)=\{B\,T\} \end{array}\right\}, \emptyset$$

CONFLICT $\quad F \approx T$

$$NOSAT$$

Fig. 3. Computation for $\{B \approx g, g\,T \approx g\,F\}$

4 The `congruence` tactic

A previous version of the algorithm has been implemented in the Coq proof assistant (version 8.0). This version is incomplete but the upcoming version 8.1 will allow you to benefit from a corrected version. The `congruence` tactic extracts equations and inequalities from the context and adds the negated conclusion if it is an equality. Otherwise, we add to the problem the inequalities $H \not\approx C$, H which is a hypotheses which is not an equation and C the conclusion. We also add $H \not\approx H'$ for every pair of hypotheses $h : H$ and $h' : \neg H'$ where neither H nor H' are equalities.

Then the algorithm is executed with a fixed strategy. If the result is SAT it fails to prove the goal. Otherwise it is instrumented to produce a constructive proof of the equation that was proven to dissatisfy the problem. The method used is similar to that explained in [4] and [9] and more recently in [12].

Moreover, the congruence tactic detects if the final set of terms is application-closed and if not, it can complete it by adding extra terms with meaningless constants, so that if the completed problem is unsatisfiable, the procedure can ask the user for a replacement to these constants. We exemplify this with the type of triples in the Coq system (type `nat`) :

```
Inductive Cube:Set :=
| Triple: nat -> nat -> nat -> Cube.
```

We now wish to prove :

```
Theorem incomplete : ∀ a b c d : nat,
  Triple a = Triple b → Triple d c = Triple d b → a = c.
```

After having introduced the hypotheses by using the `intros` command, we are able to extract the following set of equations :

$$\{\texttt{Triple a} \approx \texttt{Triple b}, \texttt{Triple d c} \approx \texttt{Triple d b}, \texttt{a} \not\approx \texttt{c}\}$$

The algorithm finds out that the set of equations is unsatisfiable, but the set of terms had to be completed to become application-closed, so it fails to produce the proof and explains it to the user :

```
Goal is solvable by congruence but some arguments are missing.
Try "congruence with (Triple a ?1 ?2) (Triple d c ?3)",
replacing metavariables by arbitrary terms.
```

The user may then give manually the additional terms to solve the goal :

```
congruence with (Triple a 0 0) (Triple d c 0).
```

5 Conclusion

The `congruence` tactic is a fast tactic which solves a precise problem, so it can be used as a goal filter, to eliminate trivial subgoals. This is very useful when doing

multiple nested case-analysis of which most cases are failure cases for example when doing the proofs for the reflexion schemes used in [3].

The work lacks both a proper complexity analysis and some benchmarks, but since it is quite specific it is difficult to compare with other tools.

An obvious improvement would be to add λ-abstraction to the term algebra. This would require us to keep track of all possible β-reductions. Then, it would be interesting to see what can be done with dependently-typed terms, since the conclusion of the CONGR$_@$ rule is ill-typed in that case. An exploration of the relation between heterogeneous equality [8] and standard homogeneous equality is needed for closed terms. Whichever extension will be done will have to keep to the original aim: design a specific procedure which can solve a problem or fail to solve it very fast so it can be used widely.

References

1. L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.
2. L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *CADE-17*, *LNCS* 1831, pages 64–78, Pittsburgh, 2000. Springer-Verlag.
3. E. Contejean and P. Corbineau. Reflecting proofs in first-order logic with equality. In *CADE-20*, *LNCS* 3632,pages 7–22, Tallinn, 2005. Springer-Verlag
4. P. Corbineau. *Autour de la clôture de congruence avec Coq*. Master's thesis, Universit Paris 7, 2001. `http://www.cs.ru.nl/~corbinea/ftp/publis/mem-dea.ps`.
5. P. Corbineau. *Dmonstration automatique en Thorie des Types*. PhD thesis, Universit Paris 11, 2005. `http://www.cs.ru.nl/~corbinea/ftp/publis/Corbineau-these.pdf`.
6. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):771–785, 1980.
7. A. A. Markov. On the impossibility of certain algorithms in the theory of associative systems. *Doklady Akademii Nauk SSSR*, 55(7):587–590, 1947. in Russian, English translation in C.R. Acad. Sci. URSS, 55, 533–586.
8. C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
9. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Mellon University, 1998. available as Technical Report CMU-CS-98-154.
10. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, October 1979.
11. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980.
12. R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *RTA'05*, *LNCS* 3467, pages 453–468, 2005. Springer-Verlag.
13. The Objective Caml language. `http://www.ocaml.org/`.
14. D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM*, 1978.
15. E. L. Post. Recursive unsolvability of a problem of Thue. *Journal of Symbolic Logic*, 13:1–11, 1947.