CC(X): Semantic Combination of Congruence Closure with Solvable Theories

Sylvain Conchon [1,2]   Evelyne Contejean [1,2]   Johannes Kanig [1,2]
Stéphane Lescuyer [1,2]

LRI, Univ. Paris-Sud, CNRS, Orsay F-91405
& INRIA Futurs, ProVal, Orsay, F-91893
FRANCE

Abstract

We present a generic congruence closure algorithm for deciding ground formulas in the combination of the theory of equality with uninterpreted symbols and an arbitrary built-in solvable theory X. Our algorithm CC(X) is reminiscent of Shostak combination: it maintains a union-find data-structure modulo X from which maximal information about implied equalities can be directly used for congruence closure. CC(X) diverges from Shostak's approach by the use of semantic values for class representatives instead of canonized terms. Using semantic values truly reflects the actual implementation of the decision procedure for X. It also enforces to entirely rebuild the algorithm since global canonization, which is at the heart of Shostak combination, is no longer feasible with semantic values. CC(X) has been implemented in Ocaml and is at the core of Ergo, a new automated theorem prover dedicated to program verification.

Keywords: decision procedures, equality theory, congruence closure, verification

1 Introduction

Combining decision procedures for the quantifier-free theory of equality over uninterpreted function symbols ($\mathcal{E}$) and other theories is at the core of a number of verification systems. For instance, problem divisions of the SMT competition [10] include the combinations of $\mathcal{E}$ and the linear arithmetics over the integers; $\mathcal{E}$ and the theory of arrays etc.

There are two main paradigms for combining theories: The Nelson-Oppen combination procedure [5] and the Shostak's algorithm [12,11]. The former procedure is very general: it applies to disjoint stably-infinite theories that communicate by an equality propagation mechanism between shared variables. However, quoting Shankar from [11], this method "has some disadvantages". Indeed, the theory $\mathcal{E}$ has no particular status in this approach and its combination amounts to implementing a specific decision procedure with the ability to infer and communicate new implied equalities, which can be very expensive.

On the contrary, Shostak's method has been specifically designed for combining $\mathcal{E}$ with (a smaller class of) solvable and canonizable theories. Again quoting Shankar [11], "Shostak's algorithm tries to gain efficiency", when it is applicable. It is based on an extension of a congruence closure algorithm that maintains a partition of terms within a table (reminiscent of a union-find data structure) mapping terms to representatives. Roughly speaking, a run of this algorithm consists in transforming equations into substitutions using *solvers*, then applying substitutions to representatives and reducing the latter to normal forms by the use of *canonizers* so that new equations can be directly drawn from the table.

A central point for Shostak method and its extensions [9] to be effective is that representatives have to be themselves *terms*. As a consequence, the main operations of the algorithm, substitution application, normal form reduction and equation resolution, have to be directly implemented on term data structures, which is not the best efficient way of implementing a decision procedure [3] (*e.g.* a term data structure is obviously not optimal to manipulate polynomials). However, relaxing this constraint has strong impacts on the design of the method. Indeed, bringing a representative into a normal form amounts to traversing its *syntactic structure* for applying the canonizers on interpreted subterms. This *global canonization* is at the heart of the method and it also guarantees the incrementality of the algorithm.

In this paper, we present an algorithm, called CC(X) (for congruence closure modulo X), which combines the theory $\mathcal{E}$ with an arbitrary built-in solvable theory X without using canonizers. This algorithm uses *abstract values* as representatives allowing efficient data structures for the implementation of solvers. CC(X) is presented as a set of inference rules whose description is low-level enough to truly reflect the actual implementation of the combination mechanism of the Ergo [1] theorem prover. Unlike Shostak's algorithm, global canonization is no longer possible with abstract values. As a consequence, incrementality of CC(X) is not obtained for free and extra rules are added in our inference system to make our algorithm incremental.

---

[3] It is also worth noting that this constraint is not imposed by the Nelson-Oppen approach.

2 Congruence Closure Modulo X

In this section, we present an extension of a congruence closure algorithm capable of combining the theory of equality with uninterpreted function symbols and another theory X that underlies certain restrictions. In the rest of this paper, Σ denotes the set of all symbols, including interpreted and uninterpreted symbols.

2.1 Solvable Theories

While solvers and canonizers of Shostak theories operate on terms directly, the theory X we are about to introduce works on a certain set $\mathcal{R}$, whose elements are called *semantic values*. The main particularity is that we don't know the exact structure of these values, only that they are somehow constructed of interpreted and uninterpreted (foreign) parts. To compensate, we dispose of two functions $[\cdot] : T(\Sigma) \to \mathcal{R}$ and *leaves* $: \mathcal{R} \to \mathcal{P}_f^*(\mathcal{R})$ which are reminiscent of the variable abstraction mechanism found in Nelson-Oppen method. $[\cdot]$ constructs a semantic value from a term; *leaves* extracts its uninterpreted parts in abstract form.

Definition 2.1 We call a *solvable theory* X a tuple $(\Sigma_X, \mathcal{R}, =_X)$, where $\Sigma_X \subseteq \Sigma$ is the set of function symbols interpreted by X, $\mathcal{R}$ is a set (of semantic values) and $=_X$ is a congruence relation *over terms*, $=_X \subseteq T(\Sigma) \times T(\Sigma)$. Additionally, a theory X has the following properties:

- There is a function $[\cdot] : T(\Sigma) \to \mathcal{R}$ to construct a semantic value out of a term. For any set E of equations between terms we write $[E]$ for the set $\{[x] = [y] \mid x = y \in E\}$.

- There is a function *leaves* $: \mathcal{R} \to \mathcal{P}_f^*(\mathcal{R})$, where the elements of $\mathcal{P}_f^*(\mathcal{R})$ are finite non-empty sets of semantic values. Its role is to return the set of maximal uninterpreted values a given semantic value consists of. Its behaviour is left undefined, but is constrained by axioms given below.

- There is a special value $\mathbf{1} \in \mathcal{R}$ which we will use to denote the leaves of pure terms' representatives.

- There is a function $subst : \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}$. Instead of $subst(p, P, r)$ we write $r\{p \mapsto P\}$. The pair (p, P) is called a *substitution* and $subst(p, P, r)$ is the *application* of a substitution to r.

- There is a (partial) function $solve : \mathcal{R} \times \mathcal{R} \to \mathcal{R} \times \mathcal{R}$.

- Let $E_{\mathcal{R}}$ be the set of equations between elements of $\mathcal{R}$. There is a relation $\models \subset \mathcal{P}(E_{\mathcal{R}}) \times \mathcal{R} \times \mathcal{R}$ whose meaning is the following: if r_1 equals r_2 can be deduced, in the model $\mathcal{R}$, from the equalities $e_1, \ldots, e_n \in E_{\mathcal{R}}$, we write

$\{e_1, \ldots, e_n\} \models r_1 = r_2$. In particular, when $\emptyset \models a = b$, this means that the semantic values a and b are equal, which we write $a \equiv b$.

In the remaining of this paper, we simply call *theory* a solvable theory. An example of such a theory is given in Section 3.

In the following, for any set S, we write S^* the set of finite sequences of elements of S. If $s \in S^*$ is such a sequence and a is an element of S, we write $a; s$ for the sequence obtained by prepending a to s. The empty sequence is denoted $\bullet$. As we will often talk about successive substitutions, we define an auxiliary function that does just that:

Definition 2.2 There is a function $iter : (\mathcal{R} \times \mathcal{R})^* \times \mathcal{R} \to \mathcal{R}$ that applies *subst* successively in the following way:

$$iter(\bullet, r) = r$$

$iter((r_1, r_2); S,\, r_3) = r_3' \{p \mapsto P\}$ where $r_i' = iter(S, r_i)$ and $(p, P) = solve(r_1', r_2')$.

In addition to definition 2.1, a theory X must fulfill the following axioms:

Axiom 2.3 For any $r_1, r_2, p, P \in \mathcal{R}$, $solve(r_1, r_2) = (p, P) \Rightarrow r_1 \{p \mapsto P\} \equiv r_2 \{p \mapsto P\}$.

Axiom 2.4 $[E] \models [u] = [v] \Rightarrow u =_{E,X} v$, where $=_{E,X}$ denotes the equational theory defined by $E \cup =_{\mathsf{x}}$.

Axiom 2.5 For any $S \in (\mathcal{R} \times \mathcal{R})^*$ and any $r \in \mathcal{R}$, we have $S \models iter(S, r) = r$ where S is seen as a set on the left-hand side of $\models$.

Axiom 2.6 For any $r, p, P \in \mathcal{R}$ such that $r \not\equiv r\{p \mapsto P\}$,

(i) $p \in leaves(r)$

(ii) $leaves(r\{p \mapsto P\}) = (leaves(r) \setminus \{p\}) \cup leaves(P)$

Axiom 2.7 For any pure term $t \in T(\Sigma_{\mathsf{X}})$, $leaves([t]) = \{\mathbf{1}\}$.

Let us explain this a little bit. First of all, as we will see in section 2.2, the algorithm establishes and maintains equivalence classes over semantic values. Every equivalence class is labeled by an element of the set $\mathcal{R}$; a function $\Delta : \mathcal{R} \to \mathcal{R}$ is maintained that for each value returns its current label. Together with the $[.]$ function, this function can be used to maintain equivalence classes over terms. The function *solve* is capable of solving an equation between two elements of $\mathcal{R}$, that is, it transforms an equation $r_1 = r_2$ for $r_1, r_2 \in \mathcal{R}$ into the substitution (p, P), with $p, P \in \mathcal{R}$, where the value p is now isolated. Axiom 2.3 makes sure that such a substitution renders equal the two semantic values r_1 and r_2, which are at the *origin* of this substitution. Finally, $\mathcal{R}$ comes also with a notion of *implication* of equalities, the relation $\models$. Axiom 2.4 just states

that, if a set $[E]$ of equations between semantic values implies an equation $[u] = [v]$, then $u =_{E,X} v$, that is, an equality on the theory side implies an equality between corresponding terms. Axiom 2.5 states that iterated substitution *iter* behaves well with respect to this implication relation: if r' has been obtained from r by iterated substitution, then the equations at the origin of these substitutions imply the equality $r' = r$ (axiom 2.5). Axiom 2.6 ensures that substituting P to p in a semantic value only has effect if p is a leaf of this value, and that the new leaves after the substitution are leaves coming from P. Finally, the last axiom describes why we introduced a special value **1** in $\mathcal{R}$: representatives of pure terms do not have leaves *per se*, but it is convenient for the algorithm that the set $leaves(r)$ be non-empty for any semantic value r. To that purpose, we arbitrarily enforce that $leaves([t])$ is the singleton $\{\mathbf{1}\}$ for any pure term t.

As a last remark, we have given the interface of a theory X in a slightly less general fashion as was possible: depending on the theory, the function *solve* may as well return a *list* of pairs (p_i, P_i) with $p_i, P_i \in \mathcal{R}$. It becomes clear why we call this a substitution: the p_i can be seen as variables, that, during the application of a substitution, are replaced by a certain semantic value. However, for the example presented in the next section, *solve* always returns a single pair, if it succeeds at all. Thus, we will stick with the simpler forms of *solve* and *subst*.

The following proposition is a simple, but useful, consequence of the axioms stated above. It will be used in the soundness proof. It simply states that, if semantic values constructed with $[\cdot]$ are equal, the original terms were already equal with respect to X.

Proposition 2.8 *For any terms* $x, y \in T(\Sigma)$, $[x] \equiv [y] \Rightarrow x =_x y$.

In order to prove the completeness, we need to make a few more assumptions about the theory X, or rather about the interpretation of symbols in Σ_X.

Axiom 2.9 For each interpreted symbol $f \in \Sigma_\mathsf{X}$ of arity n, we assume there exists a function f^X from $\mathcal{R}^n$ to $\mathcal{R}$ such that:
$$\forall t_1, \ldots, t_n \in T(\Sigma), [f(t_1, \ldots, t_n)] \equiv f^\mathsf{X}([t_1], \ldots, [t_n])$$

Note, though, that these functions need not be implemented for the algorithm to work: only their existence matters to us, [.] could be computed in any other conceivable way and our algorithm $\mathsf{CC}(\mathsf{X})$ will never need to use one of these functions explicitly. The last axiom simply state that substitutions happen at the leaves level of semantic values.

Axiom 2.10 For any interpreted symbol f, given terms $t_1, \ldots, t_n$ and two

semantic values p and P,

$$f^{\mathsf{X}}([t_1], \ldots, [t_n])\{p \mapsto P\} \equiv f^{\mathsf{X}}([t_1]\{p \mapsto P\}, \ldots, [t_n]\{p \mapsto P\})$$

2.2   *The Algorithm* CC(X)

The backtracking search underlying the architecture of a lazy SMT solver enforces an incremental treatment of the set of ground equations maintained by the solver. Indeed, for efficiency reasons, equations are given one by one by the SAT solver to decision procedures which prevents them from realizing a global preliminary treatment, unless restarting the congruence closure from scratch. Therefore, CC(X) is designed to be incremental and deals with a sequence of equations and queries instead of a given set of ground equations.

   The algorithm works on tuples (*configurations*) $\langle\, \Theta \mid \Gamma \mid \Delta \mid \Phi \,\rangle$, where:

- Θ is the set of terms already encountered by the algorithm;

- Γ is a mapping from semantic values to sets of terms which intuitively maps each semantic value to the terms that "use" it directly. This structure is reminiscent of Tarjan *et al.*'s algorithm [2] but differs in the sense that it traverses interpreted symbols (as expressed in Proposition B.6 in the appendix). This information is used to efficiently retrieve the terms which have to be considered for congruence;

- Δ a mapping from semantic values to semantic values maintaining the equivalence classes over $\mathcal{R}$ as suggested in Section 2.1: it is a structure that can tell us if two values are known to be equal (it can be seen as the *find* function of a union-find data structure);

- Φ a sequence of equations between terms that remain to be processed.

   Given a sequence E of equations and a query $a \overset{?}{=} b$ for which we want to solve the uniform word problem, CC(X) starts in an initial configuration $K_0 = \langle\, \emptyset \mid \Gamma_0 \mid \Delta_0 \mid E \,;\, a \overset{?}{=} b \,\rangle$, where $\Gamma_0(r) = \emptyset$ and $\Delta_0(r) = r$ for all $r \in \mathcal{R}$. In other words, no terms have been treated yet by the algorithm, and the partition Δ_0 corresponds to the physical equality $\equiv$.

   In Figure 1, we describe our algorithm CC(X) as four inference rules operating on configurations. The semantic value $\Delta(r)$, for $r \in \mathcal{R}$ is also called *representative* of r. When t is a term of $T(\Sigma)$, we write $\Delta[t]$ as an abbreviation for $\Delta([t])$, which we call the representative of t. Figure 1 also uses several other abbreviations: we write $\vec{u}$ for $u_1, \ldots, u_n$, where n is clear from the context ; we also write $\Delta[\vec{u}] \equiv \Delta[\vec{v}]$ for the equivalences $\Delta[u_1] \equiv \Delta[v_1], \ldots, \Delta[u_n] \equiv \Delta[v_n]$. If $t \in \Gamma(r)$ for $t \in T(\Sigma), r \in \mathcal{R}$, we also say r is *used by* t, or t *uses* r.

   We now have all the necessary elements to understand the rules. There are actually only two of them, namely CONGR and ADD, which perform any

$$\text{CONGR} \frac{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ a = b\ ;\ \Phi\ \ \rangle}{\langle\ \ \Theta\ \ |\ \ \Gamma \uplus \Gamma'\ \ |\ \ \Delta'\ \ |\ \ \Phi'\ ;\ \Phi\ \ \rangle}\ a, b \in \Theta,\ \ \Delta[a] \not\equiv \Delta[b]$$

where,

$$(p, P) = solve(\Delta[a], \Delta[b])$$

$$\Gamma' = \bigcup_{l \in leaves(P)} l \mapsto \Gamma(l) \cup \Gamma(p)$$

$$\forall r \in \mathcal{R}, \Delta'(r) := \Delta(r)\{p \mapsto P\}$$

$$\Phi' = \left\{ f(\vec{u}) = f(\vec{v}) \left| \begin{array}{l} \Delta'[\vec{u}] \equiv \Delta'[\vec{v}],\ \ f(\vec{u}) \in \Gamma(p) \\ f(\vec{v}) \in \Gamma(p) \cup \bigcup_{t \in \Theta | p \in leaves(\Delta[t])} \bigcap_{l \in leaves(\Delta'[t])} \Gamma(l) \end{array} \right. \right\}$$

$$\text{REMOVE} \frac{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ a = b\ ;\ \Phi\ \ \rangle}{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ \Phi\ \ \rangle}\ a, b \in \Theta,\ \ \Delta[a] \equiv \Delta[b]$$

$$\text{ADD} \frac{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ C[f(\vec{a})]\ ;\ \Phi\ \ \rangle}{\langle\ \ \Theta \cup \{f(\vec{a})\}\ \ |\ \ \Gamma \uplus \Gamma'\ \ |\ \ \Delta\ \ |\ \ \Phi'\ ;\ C[f(\vec{a})]\ ;\ \Phi\ \ \rangle}\ \left\{ \begin{array}{l} f(\vec{a}) \notin \Theta \\ \forall v \in \vec{a}, v \in \Theta \end{array} \right.$$

where $C[f(\vec{a})]$ denotes an equation or a query containing the term $f(\vec{a})$

$$\text{with}\ \left\{ \begin{array}{l} \Gamma' = \bigcup_{l \in \mathcal{L}_\Delta(\vec{a})} l \mapsto \Gamma(l) \cup \{f(\vec{a})\} \\[2ex] \Phi' = \left\{ f(\vec{a}) = f(\vec{b}) \left| \Delta[\vec{a}] \equiv \Delta[\vec{b}],\ \ f(\vec{b}) \in \bigcap_{l \in \mathcal{L}_\Delta(\vec{a})} \Gamma(l) \right. \right\} \end{array} \right.$$

where $\mathcal{L}_\Delta(\vec{a}) = \bigcup_{v \in \vec{a}} leaves(\Delta[v])$

$$\text{QUERY} \frac{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ a \stackrel{?}{=} b\ ;\ \Phi\ \ \rangle}{\langle\ \ \Theta\ \ |\ \ \Gamma\ \ |\ \ \Delta\ \ |\ \ \Phi\ \ \rangle}\ a, b \in \Theta,\ \ \Delta[a] \equiv \Delta[b]$$

Fig. 1. The rules of the congruence closure algorithm CC(X)

interesting tasks. The other two are much simpler: REMOVE just checks if the first equation in Φ is already known to be true (by the help of Δ), and, if so, discards it. The QUERY rule is analogous to the REMOVE rule but deals with a query.

The rule CONGR is more complex. It also inspects the first equation in Φ, but only when it is not already known to be true. This equation $a = b$ with $a, b \in \Theta$ is transformed into an equation in $\mathcal{R}$, $\Delta[a] = \Delta[b]$, and then solved in the theory X, which yields two semantic values p and P. The value p is then substituted by P in all representatives. The map Γ is updated according to this substitution: the terms that used p before now also use all the values $l \in leaves(P)$. Finally, a set Φ' of new equations is calculated, and appended to the sequence Φ of the equations to be treated

(the order of the equations in Φ' is irrelevant). The set Φ' is computed in the following way: the left hand side of any equation in Φ' is a term that used p, and the right hand side is either a term that used p, or a term that used every $l \in leaves(\Delta'(r))$ for a value r such that $p \in leaves(\Delta(r))$. This rather complicated condition ensures that only relevant terms are considered for congruence. As the name implies, the CONGR rule will only add equations of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$, where the corresponding subterms are already known to be equal: $\Delta'[t_i] \equiv \Delta'[t'_i], 1 \leq i \leq n$.

The rule ADD is used when the first equation of Φ contains at least a term $f(\vec{a})$ that has not yet been encountered by the algorithm ($f(\vec{a}) \notin \Theta$). Its side condition ensures that all proper subterms of this term have been added before ; in other words, new terms are added recursively. The first task that this rule performs is of course to update the map Γ by adding the information that $f(\vec{a})$ uses all the leaves of its direct subterms. However, this is not sufficient: we lose the completeness of the algorithm if no equation is added during the application of an ADD rule. Indeed, suppose for instance that Φ is the sequence $f(a) = t; a = b; f(b) = u$. Then, we would fail to prove that $t = u$ since the equality $a = b$ is processed too early. At this point, $f(b)$ has not been added yet to the structure Γ, thus preventing the congruence equation $f(a) = f(b)$ to be discovered in the CONGR rule. For this reason, the ADD rule also performs congruence closure by looking for equations involving the new term $f(\vec{a})$: this is the construction of the set Φ' of equations, where the restrictive side condition over $f(\vec{b})$ ensures that only relevant terms are considered.

Soundness and completeness proofs of $\mathsf{CC}(\mathsf{X})$ are given in appendices A and B. Since no new terms are generated during $\mathsf{CC}(\mathsf{X})$'s execution, the number of potential equations to be handled is quadratically bounded by the input problem size.

3   Example

In this section, we present the theory $\mathcal{A}$ of linear arithmetic over the rationals as an interesting example of instantiation of $\mathsf{CC}(\mathsf{X})$. This theory consists of the following elements:

- The interpreted function symbols are $+, -, /, \times, succ$ and 0.
- The semantic values are polynomials of the form

$$c_0 + \sum_{i=1}^{n} c_i \boxed{r_i}, \quad c_i \in \mathbb{Q}, \boxed{r_i} \in T(\Sigma), c_i \neq 0.$$

From an implementation point of view, these polynomials can be represented as pairs where the left component represents c_0 and the right component is a map from foreign values (not handled by linear arithmetic; these are surrounded by a box in this example) to rationals that represents the sum $\sum_{i=1}^{n} c_i \boxed{r_i}$. Note that in semantic values, $+$ is *not* the interpreted function symbol but just notation to separate the different components of the polynomial.

- $=_A$ is just the usual equality of linear arithmetic over rationals.

  The functions needed by the algorithm are defined as follows:

- The function $[\cdot]$ interprets the above function symbols as usual and constructs polynomials accordingly.

- The function leaves just returns the set of all the foreign values in the polynomial:

$$leaves \left(c_0 + \sum_{i=1}^{n} c_i \boxed{r_i} \right) = \{ \boxed{r_i} \mid 1 \leq i \leq n \}.$$

- For the value $\boxed{r}$ and the polynomials p_1, p_2, $subst(\boxed{r}, p_1, p_2)$ replaces the foreign value $\boxed{r}$ by the polynomial p_1 in p_2, if r occurs in p_2.

- For two polynomials $p_1, p_2 \in \mathcal{R}$, $solve(p_1, p_2)$ is simply the Gauss algorithm that solves the equation $p_1 = p_2$ for a certain foreign values occurring in p_1 or p_2.

- $\models$ is again just the entailment relation in linear arithmetic.

  If we admit the soundness of the $[\cdot]$ function and the Gauss algorithm used in *solve*, the axioms that need to hold are trivially true.

Theorem 3.1 *The functions defined above satisfy the axioms 2.3 - 2.5.*

We now want to show the execution of the non-incremental algorithm by an example in arithmetic. Consider therefore the set of equations

$$E = \{ g(x + k) = a, s = g(k), x = 0 \}$$

and we want to find out if the equation $s = a$ follows. The algorithm starts in the initial configuration $K_0 = \langle \, \emptyset \mid \Gamma_0 \mid \Delta_0 \mid E \; ; \; s \stackrel{?}{=} a \, \rangle$, as defined in section 2.2. In the following, components of the configuration with the subscript i denote the state of the component after complete treatment of the ith equation.

Before being able to treat the first equation $g(x + k) = a$ using the CONGR rule, all the terms that appear in the equation have to be added by the ADD rule. This means in particular that the components Γ and Θ are updated according to Fig. 1. No new equations are discovered, so Φ and Δ remain unchanged. Now we can apply the CONGR rule to the first equation $g(x +$

$k) = a$. This yields an update of Γ and Δ, but no congruence equations are discovered. This is the configuration after the treatment of the first equation:

$$\Gamma_1 = \left\{ \boxed{x} \mapsto \{x + k,\ g(x + k)\},\ \boxed{k} \mapsto \{x + k,\ g(x + k)\} \right\}\ \cup\ \Gamma_0$$
$$\Delta_1 = \left\{ \boxed{g(x + k)} \mapsto \boxed{a},\ \boxed{a} \mapsto \boxed{a} \right\}\ \cup\ \Delta_0$$

The second equation is treated similarly: The terms s and $g(k)$ are ADDED and the representative of $g(k)$ becomes $\boxed{s}$. These are the changes to the structures Γ and Δ:

$$\Gamma_2 = \left\{ \boxed{k} \mapsto \{x + k,\ g(x + k),\ g(k)\} \right\}\ \cup\ \Gamma_1$$
$$\Delta_2 = \left\{ \boxed{g(k)} \mapsto \boxed{s},\ \boxed{s} \mapsto \boxed{s} \right\}\ \cup\ \Delta_1$$

The most interesting part is the treatment of the third equation, $x = 0$, because we expect the equation $g(x + k) = g(k)$ to be discovered. Otherwise, the algorithm would be incomplete. Every term in the third equation has already been added, so we can directly apply the CONGR rule. $solve(\Delta_2\,[a]\,, \Delta_2\,[b])$ returns the substitution $(x, 0)$, which is applied to all representatives. The value 0 is a pure arithmetic term, so $leaves(0)$ returns $\{\mathbf{1}\}$. We obtain the following changes to Γ_3 and Δ_3:

$$\Gamma_3 = \{\mathbf{1} \mapsto \{x + k,\ g(x + k)\}\}\ \cup\ \Gamma_2$$
$$\Delta_3 = \left\{ \boxed{x} \mapsto 0, \boxed{x} + \boxed{k} \mapsto \boxed{k} \right\}\ \cup\ \Delta_2$$

It is important to see that the representative of $x + k$ has changed, even if the term was not directly involved in the equation that was treated.

To discover new equations, the set Φ_3 has to be calculated. To calculate this set, we first collect the terms that use x:

$$\Gamma_2(\boxed{x}) = \{x + k, g(x + k)\}.$$

The elements of $\Gamma_2(x)$ are potential lhs of new equations. To calculate the set of potential rhs, we first construct the set of values r corresponding to terms in Θ_2 such that the representative of r contains x:

$$\{r \mid x \in leaves(\Delta_2(r))\} = \left\{ \boxed{x}, \boxed{x} + \boxed{k} \right\}$$

Now, for every value r in this set, we calculate $leaves(\Delta_3(r))$ and construct their intersection:

$$\bigcap_{l \in leaves(0)} \Gamma_2(l) = \Gamma_2(\mathbf{1}) = \emptyset$$

$$\bigcap_{l \in leaves\left(\boxed{k}\right)} \Gamma_2(l) = \{x + k, g(x + k), g(k)\}$$

The union of the two sets and the set $\Gamma_2(\boxed{x})$ is the set of potential rhs $\{x + k, g(x + k), g(k)\}$. If we cross this set with the set $\Gamma_2(\boxed{x})$ and filter the equations that are not congruent, we obtain three new equalities

$$\Phi_3 = x + k = x + k \; ; \; g(x + k) = g(x + k) \; ; \; g(x + k) = g(k) \; ; \; s \stackrel{?}{=} a.$$

The first two equations get immediately removed by the REMOVE rule. The third one, by transitivity, delivers the desired equality which permits to discharge the query $s \stackrel{?}{=} a$.

4 Implementation

An efficient Ocaml implementation of $\mathsf{CC}(\mathsf{X})$ exists and is at the heart of the Ergo automated theorem prover [1]. This implementation uses only purely functional data-structures and directly follows the inference rules presented in Section 1.

In order to check the scalability of our algorithm, we benchmarked Ergo on our test suite: 1450 verification conditions automatically generated by the VCG Caduceus/Why from 69 C programs [8]. These goals make heavy use of equalities over uninterpreted function symbols and linear arithmetic. Figure 2 shows the results of the comparison between Ergo and four other provers: Z3, Yices, Simplify and CVC3. The five provers were run with a fixed timeout of 20s on a machine with Xeon processors (3.20 GHz) and 2 Gb of memory. For this benchmark, $\mathsf{CC}(\mathsf{X})$ is instantiated with the theory of linear arithmetic.

	valid	timeout	unknown	avg. time
Simplify v1.5.4	98%	1%	1%	60ms
Yices v1.0	95%	2%	3%	210ms
Ergo v0.7	94%	5%	1%	150ms
Z3 v0.1	87%	10%	3%	690ms
CVC3 v20070307	71%	1%	28%	80ms

Fig. 2. Comparison between Ergo, Simplify, Yices, CVC3 and Z3 on 1450 verification conditions.

The column **valid** shows the percentage of the conditions proved valid by the provers [4]. The column **timeout** gives the percentage of timeouts whereas **unknown** shows the amount of problems unsolved due to incompleteness. Finally, the column **avg. time** gives the average time for giving a valid answer.

As shown by the results in Figure 2, the current experimentations are very promising with respect to speed and to the number of goals automatically

---

[4] All conditions of our test suite are proved valid at least by one prover.

solved. However, the benchmarks also contain logical connectives and quantifiers, not handled by CC(X). So, strictly speaking, Figure 2 only proves that CC(X) is sufficiently fast to let Ergo compete with state-of-the-art SMT solvers.

5   Conclusion and Future Work

We have presented a new algorithm CC(X) which combines the theory of equality over uninterpreted function symbols with a solvable theory. Our method is reminiscent of Shostak's algorithm [12,11,3]. Its main novelty rests on the use of abstract data structures for class representatives that allows efficient implementations of crucial operations. Our approach is also higly modular: contrarily to *ad-hoc* extensions of congruence closure [6,7], CC(X) can be instantiated with an arbitrary solvable theory underlying the restrictions described in Section 2.1.

   CC(X) has been implemented in Ocaml as a functor parameterized by a theory module whose signature is the one given in section 2.1. CC(X) is at the core of the Ergo theorem prover. Since practice often arrives before theory, a number of extra features of CC(X) have already been implemented in Ergo. We leave for future work their formalization and correctness proofs:

- A functor CombineX(X1,X2) combines two theory modules X1 and X2, allowing CC(X) to combine several solvable theories. As shown in [4], solvers for first order theories almost never combine. However, while this is out of the scope of this paper, we claim that solvers for *typed theories* (under certain restrictions) can be combined.

- Predicate symbols are already handled by CC(X). Their treatment smoothly integrates to the overall framework.

- CC(X) has been instrumented to produce explications so that the SAT solver part of Ergo can benefit from them for its backjumping mechanism.

   Another direction is to "prove the prover" in a proof assistant. Indeed, Ergo uses only purely functional data-structures, is highly modular and very concise ($\sim$ 3000 lines of code). All these features should make a formal certification feasible.

References

[1] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[2] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *J. ACM*, 27(4):771–785, 1980.

[3] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2001.

[4] S. Krstić and S. Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2):87–106, May 2005.

[5] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, Oct. 1979.

[6] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[7] R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007.

[8] ProVal Project. Why Benchmarks. http://proval.lri.fr/why-benchmarks/.

[9] S. Ranise, C. Ringeissen, and D. K. Tran. Nelson-oppen, shostak and the extended canonizer: A family picture with a newborn. In *ICTAC*, pages 372–386, 2004.

[10] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2006. http://www.smtcomp.org.

[11] H. Rueß and N. Shankar. Deconstructing Shostak. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 19, Washington, DC, USA, 2001. IEEE Computer Society.

[12] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31:1–12, 1984.

A   Soundness Proof of CC(X)

We now proceed to prove the soundness of the algorithm. Let E be a set of equations between terms of $T(\Sigma)$ and X a theory in the sense of Definition 2.1. For the proof, we need an additional information about the run of an algorithm, that is not contained in a configuration: the set O of equations that have already been treated in a CONGR rule.

The first proposition shows that the equations that are already treated are never contradicted by Δ.

Proposition A.1 *For any configuration* $\langle\ \Theta\ |\ \Gamma\ |\ \Delta\ |\ \Phi\ \rangle$ *and for all* $t_1, t_2 \in T(\Sigma)$ *we have:* $t_1 = t_2 \in O \Rightarrow \Delta[t_1] \equiv \Delta[t_2]$.

The next proposition shows that Δ coincides with the function *iter*, applied to the equations that have already been treated.

Proposition A.2 *For any configuration* $\langle\ \Theta\ |\ \Gamma\ |\ \Delta\ |\ \Phi\ \rangle$ *and for all* $t \in T(\Sigma)$ *we have:* $\Delta[t] = iter([O], [t])$.

The next proposition states that the evolution of the representative of a term is always justified by the equations that have been treated:

Proposition A.3 *For any configuration* $\langle\ \Theta\ |\ \Gamma\ |\ \Delta\ |\ \Phi\ \rangle$ *and for all* $t \in T(\Sigma)$ *we have:* $[O] \models \Delta_0[t] = \Delta[t]$.

This is the main lemma: It basically states the soundness of Δ, crucial for the soundness of the whole algorithm.

Lemma A.4 *For any configuration* $\langle\,\Theta\mid\Gamma\mid\Delta\mid\Phi\,\rangle$ *and for all* $t_1, t_2 \in T(\Sigma)$, *we have:*

$$\Delta[t_1] \equiv \Delta[t_2] \Rightarrow t_1 =_{X,O} t_2.$$

We are now ready to state the main soundness theorem: whenever two terms have the same representative, they are equal w.r.t. the equational theory defined E and X, and every newly added equation is sound as well. For the soundness of the algorithm, we are only interested in the first statement, but we need the second to prove the first, and the statements have to be proved in parallel by induction.

Theorem A.5 *For any configuration* $\langle\,\Theta\mid\Gamma\mid\Delta\mid\Phi\,\rangle$, *we have:*

$$\forall t_1, t_2 \in T(\Sigma): \quad \Delta[t_1] \equiv \Delta[t_2] \Rightarrow t_1 =_{X,E} t_2$$
$$\forall t_1, t_2 \in T(\Sigma): \quad t_1 = t_2 \in \Phi \Rightarrow t_1 =_{X,E} t_2.$$

Proof. We prove the two claims simultaneously by induction on the application of the rules CONGR, REMOVE, ADD and QUERY. First, we observe that both claims are true for the initial configuration K_0: The second claim is trivial as $\Phi = E$, and the first claim is true because of proposition 2.8.

In the induction step, consider the last rule applied to the configuration $\langle\,\Theta\mid\Gamma\mid\Delta\mid\Phi\,\rangle$, and show that the claims still hold in the configuration obtained by application of that rule. For the rules REMOVE and QUERY this is actually trivial, as Δ does not change and Φ does not get any new equalities added. For the rule ADD, the first claim is trivial, as Δ remains unchanged. The second claim is established as follows. If $t_1 = t_2 \in \Phi$, we can conclude by induction hypothesis. If $t_1 = t_2 \in \Phi'$, then $t_1 \equiv f(\vec{a})$ and $t_2 \equiv f(\vec{b})$, for f with arity n. The conditions in figure 1 guarantee that $\Delta[\vec{a}] \equiv \Delta[\vec{b}]$. By the first claim, we can state that $a_i =_{X,E} b_i$ $(1 \leq i \leq n)$ and by the congruence property of $=_{X,E}$ we have $f(\vec{a}) =_{X,E} f(\vec{b})$, which proves the second claim.

We finally assume that the last rule applied was a CONGR rule. To prove the first claim, we assume $\Delta'[t_1] \equiv \Delta'[t_2]$. By lemma A.4, we have $t_1 =_{X,O,a=b} t_2$. Now, $a = b$ is obviously an element of the set $\{a = b\} \cup \Phi$, so that, by induction hypothesis, $a =_{X,E} b$. By the induction hypothesis and proposition A.1, for any $a_i = b_i \in O$ we have also $a_i =_{X,E} b_i$. As $=_{X,E}$ is a congruence relation, we can conclude $t_1 =_{X,E} t_2$. The second claim can be proved as in the case of the ADD rule, by the aid of the first claim. $\qquad\square$

B   Completeness Proof of CC(X)

We finally proceed to the completeness of the algorithm. In opposition to the correctness proof, we are now interested in the fact that every possible equation on the terms of the problem can be deduced by the algorithm, and in particular we are interested in its termination.

B.1   Termination and congruence closure of Δ

In the following, we assume the set E and the query $a \overset{?}{=} b$ are fixed and we denote the successive configurations by $\langle\, \Theta_n \mid \Gamma_n \mid \Delta_n \mid \Phi_n \,\rangle$ with $n = 0$ the initial configuration (as defined in Section 2.2). Let T_Π be the set of terms and subterms that appear in $E; a \overset{?}{=} b$, in particular, T_Π is closed by subterm. At any stage n in the algorithm, we write O_n for the set of equations that have been treated by the algorithm so far through the rule CONGR.

The first property we are interested in is the fact that all the equations inferred, and thus all the terms added, are only using terms from T_Π.

Proposition B.1 *For any n, $Im(\Gamma_n) \subseteq T_\Pi$, $\Phi_n \subseteq T_\Pi \times T_\Pi$ and $\Theta_n \subseteq T_\Pi$.*

Theorem B.2 (Termination) *The algorithm terminates on any input problem Π.*

Proof. To prove that this system terminates, it is sufficient to consider the measure defined as $(|T_\Pi \setminus \Theta_n|, |\Delta_n / \equiv|, |\Phi_n|)$, where the second component represents the number of equivalence classes over T_Π in Δ_n. In particular, Proposition B.1 ensures that all the terms added are in T_Π, and thus that the first component of the measure decreases through the rule ADD.

$\square$

Now, we know there exists a final configuration, for $n = \omega$. At this stage, all the equations from the original problem have been treated, and every term in T_Π has been encountered :

Proposition B.3 $O_\omega \supseteq E$.

Corollary B.4 *At the end of the algorithm, $\Theta_\omega = T_\Pi$.*

Proposition B.5 *The function $n \mapsto \Gamma_n$ is nondecreasing, i.e. $\Gamma_n(r) \subseteq \Gamma_{n+1}(r)$ for all r and n.*

The following proposition gives the true "meaning" of the map Γ_n. It shows that a term in Θ_n uses all the leaves of the representatives of its direct subterms.

Proposition B.6 *For any term $f(t_1, \ldots, t_m)$ in Θ_n, if there exists $i \leq m$ such that $p \in leaves(\Delta_n[t_i])$, then $f(t_1, \ldots, t_n) \in \Gamma_n(p)$.*

Proof. The proof proceeds by induction on n. The result holds trivially for the initial configuration since Θ_0 is empty. If the result holds after n steps, we proceed by case analysis on the rule used to get to the $n + 1$-th step. The rules REMOVE, QUERY do not change Θ_n, Γ_n or Δ_n, so if one of these rules is used the result still holds at $n + 1$. We detail both remaining rules :

CONGR: Let $f(t_1, \ldots, t_m) \in \Theta_{n+1} = \Theta_n$, and i and p such that $p \in leaves(\Delta_{n+1}[t_i])$. If (v, R) is the substitution applied, by definition of Δ_{n+1}, $p \in leaves(\Delta_n[t_i]\{v \mapsto R\})$. Now, we distinguish two cases :
  • if $p \in leaves(\Delta_n[t_i])$, then by induction hypothesis, we know that $f(t_1, \ldots, t_n) \in \Gamma_n(p)$, and thus $f(t_1, \ldots, t_n) \in \Gamma_{n+1}(p)$ by B.5.
  • if $p \notin leaves(\Delta_n[t_i])$, then $\Delta_n[t_i]$ has been changed by the substitution and the axiom 2.6 tells us that $v \in leaves(\Delta_n[t_i])$ and $p \in leaves(R)$. Therefore, by applying the induction hypothesis to v and the definition of Γ_{n+1}, we can conclude that :
$$f(t_1, \ldots, t_n) \in \Gamma_n(v) \subseteq \Gamma_n(p) \cup \Gamma_n(v) = \Gamma_{n+1}(p)$$

ADD: If $f(t_1, \ldots, t_m)$ was already in Θ_n, then it is straightforward to check that for all $p \in leaves(\Delta_{n+1}([t_i]))$, p was already in $\Delta_n[t_i]$ and the induction hypothesis together with the monotonicity of Γ_n gives us the wanted result.

If $f(t_1, \ldots, t_m)$ is in fact the new term $f(\vec{a})$ added by the rule, then let $p \in leaves(\Delta_{n+1}[t_i])$. Again, p was already in $\Delta_n[t_i]$ and since t_i is a direct subterm of the new added term $f(\vec{a})$, we have by definition that $f(\vec{a}) \in \Gamma_{n+1}(p) = \Gamma_n(p) \cup \{f(\vec{a})\}$. □

The next proposition is the central property ensuring the completeness of the algorithm, and states that Δ_ω indeed represents a congruence relation.

Proposition B.7 *The restriction of Δ_ω to T_Π is congruence-closed, i.e.*
$$\forall f(\vec{a}), f(\vec{b}) \in T_\Pi, \ \Delta_\omega[\vec{a}] \equiv \Delta_\omega[\vec{b}] \ \Rightarrow \ \Delta_\omega[f(\vec{a})] \equiv \Delta_\omega[f(\vec{b})].$$

Proof. Let k the smallest integer such that both $f(\vec{a})$ and $f(\vec{b})$ belong to Θ_k. Because terms can only be added to Θ by the rule ADD, we know the rule applied at the previous step was ADD. We can safely assume the term added was $f(\vec{a})$, by switching $\vec{a}$ and $\vec{b}$ if necessary. If $f(\vec{a})$ and $f(\vec{b})$ are equal, the result is obvious. Otherwise, $f(\vec{a}) \neq f(\vec{b})$ and $f(\vec{b})$ had been added before and was in Θ_{k-1}. Now there are two cases, depending on whether $\Delta_{k-1}[\vec{a}] \equiv \Delta_{k-1}[\vec{b}]$ or not.

• if $\Delta_{k-1}[\vec{a}] \equiv \Delta_{k-1}[\vec{b}]$, we will prove that $f(\vec{a}) = f(\vec{b})$ has been added to Φ_k, that is to say we need to establish that : $\forall i, \forall l \in leaves(\Delta_{k-1}[a_i])$, $f(\vec{b}) \in \Gamma_{k-1}(l)$. For any such i and l, we know that l is in $leaves(\Delta_{k-1}[a_i])$, and therefore in $leaves(\Delta_{k-1}[b_i])$. By Proposition B.6, this means that $f(\vec{b}) \in$

$\Gamma_{k-1}(l)$, which is exactly what we wanted.

- if on the contrary, $[\vec{a}]$ and $[\vec{b}]$ were not equal in Δ_{k-1}, then let $j \geq k$ be the smallest integer such that $\Delta_j[\vec{a}] \equiv \Delta_j[\vec{b}]$. The rule applied at the previous step must be CONGR since only CONGR changes Δ. Thus, a substitution $\{p \mapsto P\}$ has made $\Delta_{j-1}[\vec{a}]$ and $\Delta_{j-1}[\vec{b}]$ equal: there exists an i, such that

$$\Delta_{j-1}[a_i] \not\equiv \Delta_{j-1}[b_i] \ \wedge \ \Delta_{j-1}[a_i]\{p \mapsto P\} \equiv \Delta_{j-1}[b_i]\{p \mapsto P\}.$$

This means that at least one of these values, say $\Delta_{j-1}[a_i]$, has been changed by the substitution and by Axiom 2.6, that $p \in leaves(\Delta_{j-1}[a_i])$. Proposition B.6 ensures that $f(\vec{a}) \in \Gamma_{j-1}(p)$.

We still have to prove that $f(\vec{b})$ verifies the conditions in the rule CONGR, namely that $f(\vec{b}) \in \Gamma_{j-1}(p) \cup \bigcup_{t | p \in leaves(\Delta_{j-1}(t))} \bigcap_{l \in leaves(\Delta_j(t))} \Gamma_{j-1}(l)$. Again, we distinguish two cases :

· if $\Delta_{j-1}[b_i] \not\equiv \Delta_j[b_i]$, then by the same argument as above for $f(\vec{a})$, $f(\vec{b}) \in \Gamma_{j-1}(p)$ and $f(\vec{b})$ has the desired property.

· if $\Delta_{j-1}[b_i] \equiv \Delta_j[b_i]$, then $leaves(\Delta_j[a_i]) = leaves(\Delta_j[b_i]) = leaves(\Delta_{j-1}[b_i])$ and by applying Proposition B.6 once again, we deduce that for every l in $leaves(\Delta_j[a_i])$, $f(\vec{b}) \in \Gamma_{j-1}(l)$. Since $p \in leaves(\Delta_{j-1}[a_i])$, this means indeed that :

$$f(\vec{b}) \in \bigcup_{t | p \in leaves(\Delta_{j-1}(t))} \bigcap_{l \in leaves(\Delta_j(t))} \Gamma_{j-1}(l).$$

So far, we have established that the equation $f(\vec{a}) = f(\vec{b})$ has been added when the rule CONGR was applied at the step $j-1$, and thus that $f(\vec{a}) = f(\vec{b})$ belongs to Φ_j. At the end of the algorithm, this equation must have been treated. Thus, by A.1, we know that the representatives of $f(\vec{a})$ and $(\vec{b})$ are equal in Δ_ω.

$\square$

The axioms 2.9 and 2.10 introduced in Section 2.1 are used to prove that the Δ_ω component of the final configuration is coherent with the theory X, that is to say :

Proposition B.8 *Let $f(t_1, \ldots, t_n)$ a term in T_Π where f is an interpreted symbol. Then, $\Delta_\omega[f(t_1, \ldots, t_n)] \equiv f^X(\Delta_\omega[t_1], \ldots, \Delta_\omega[t_n])$.*

Proof. We will prove this result by proving it (by simple induction) for Δ_n for every N between 0 and the final configuration.

First, we observe that the result is true for the initial configuration, ie. $\Delta_0[f(t_1, \ldots, t_m)] \equiv f^X(\Delta_0[t_1], \ldots, \Delta_0[t_m])$ because it directly follows from Axiom 2.9 and the definition of Δ_0.

Now, it is sufficient to show that if the equality holds for Δ_n, it still holds in Δ_{n+1}. Since the only rule that changes Δ_n where it is already defined is

CONGR, the result is obvious for any other rule. In the case of a CONGR rule, let p, P be the substitution applied to Δ_n :

$$
\begin{aligned}
\Delta_{n+1}[f(t_1, \ldots, t_m)] &= \Delta_n[f(t_1, \ldots, t_m)]\{p \mapsto P\} \text{ by definition} \\
&\equiv f^{\mathsf{X}}(\Delta_n[t_1], \ldots, \Delta_n[t_m])\{p \mapsto P\} \text{ by induction} \\
&\equiv f^{\mathsf{X}}(\Delta_n[t_1]\{p \mapsto P\}, \ldots, \Delta_n[t_m]\{p \mapsto P\}) \text{ by } 2.10 \\
&\equiv f^{\mathsf{X}}(\Delta_{n+1}[t_1], \ldots, \Delta_{n+1}[t_m]) \text{ by definition}
\end{aligned}
$$

which proves the result.   □

In other words, this property means that Δ actually represents a union-find structure modulo X, that is, it behaves correctly with respect to the interpreted symbols.

B.2   Completeness

The completeness expresses the fact that if the query is entailed by the set of equations E and the theory X, it is proved true by $\mathsf{CC}(\mathsf{X})$. In other words, using standard model-theoretic notations, we need to prove that:

$$
E, =_{\mathsf{X}} \models a = b \quad \Rightarrow \quad \Delta_\omega[a] \equiv \Delta_\omega[b].
$$

The first step of the proof is to build a Σ-structure $\mathcal{M}$ which models E and $=_{\mathsf{X}}$, and such that the interpretation in $\mathcal{M}$ coincides with Δ_ω on $[a]$ and $[b]$.

Definition B.9 Let $\mathcal{M}$ be the structure defined in the following way :

- the domain of $\mathcal{M}$ is the set $\mathcal{R}$ of semantic values
- for each symbol $f \in \Sigma$ of arity n, we distinguish whether f is interpreted in X or not :
  · if $f \in \Sigma_{\mathsf{X}}$, then $f^{\mathcal{M}} \stackrel{def}{=} f^{\mathsf{X}}$
  · if $f \notin \Sigma_{\mathsf{X}}$, and $r_1, \ldots, r_n \in \mathcal{R}$, then the idea is to use Δ_ω wherever we can :

$$
\begin{aligned}
f^{\mathcal{M}}(r_1, \ldots, r_n) &\stackrel{def}{=} \Delta_\omega[f(t_1, \ldots, t_n)] \quad \text{if } f(t_1, \ldots, t_n) \in T_\Pi \\
&\qquad\qquad\qquad\qquad\qquad \text{and } \forall i, r_i \equiv \Delta_\omega[t_i],
\end{aligned}
$$

$$
f^{\mathcal{M}}(r_1, \ldots, r_n) \stackrel{def}{=} \mathbf{1} \qquad\qquad\qquad\qquad \text{otherwise.}
$$

  Here, we use $\mathbf{1}$, but we could use any element in $\mathcal{R}$, since we will see that it does not matter how we define interpretations in this case.

Proof. The very first thing we have to do is to prove that the definition we just gave is indeed a definition. In the case where $f^{\mathcal{M}}$ is defined in terms of Δ_ω, there may be several ways to pick the terms t_i and we have to show that the result does not depend on this choice. Let $t_1, \ldots, t_n, u_1, \ldots, u_n$ be terms

such that $\Delta_\omega[t_i] \equiv r_i \equiv \Delta_\omega[u_i]$ for all i. By Proposition B.7, we know that $\Delta_\omega[f(t_1,\ldots,t_n)] \equiv \Delta_\omega[f(u_1,\ldots,u_n)]$, which means exactly that the definition of $f^\mathcal{M}(r_1,\ldots,r_n)$ does not depend on the choice of the t_i.   □

Now that $\mathcal{M}$ is a well-defined Σ-structure, we will first show that on all the terms in T_Π, the interpretation in $\mathcal{M}$ is exactly the function $\Delta_\omega[.]$.

Lemma B.10 *For any term* $t \in T_\Pi$, $\mathcal{M}(t) \equiv \Delta_\omega[t]$.

Proof. We proceed by structural induction on terms.

Let $t = f(t_1,\ldots,t_n) \in T_\Pi$, we can apply the induction hypothesis to all the t_i because T_Π is closed by subterm. Thus, for all i, $\mathcal{M}(t_i) \equiv \Delta_\omega[t_i]$.

Now, if $f \notin \Sigma_\mathsf{X}$,

$$\mathcal{M}(f(t_1,\ldots,t_n)) = f^\mathcal{M}(\mathcal{M}(t_1),\ldots,\mathcal{M}(t_n))$$
$$\equiv f^\mathcal{M}(\Delta_\omega[t_1],\ldots,\Delta_\omega[t_n]) \text{ by IH}$$
$$\equiv \Delta_\omega[f(t_1,\ldots,t_n)] \text{ by definition of } f^\mathcal{M} \text{ for } f \notin \Sigma_\mathsf{X}$$

If $f \in \Sigma_X$, then

$$\mathcal{M}(f(t_1,\ldots,t_n)) = f^\mathcal{M}(\mathcal{M}(t_1),\ldots,\mathcal{M}(t_n))$$
$$\equiv f^\mathcal{M}(\Delta_\omega[t_1],\ldots,\Delta_\omega[t_n]) \text{ by IH}$$
$$\equiv f^\mathsf{X}(\Delta_\omega[t_1],\ldots,\Delta_\omega[t_n]) \text{ by definition of } f^\mathcal{M}$$
$$\equiv \Delta_\omega[f(t_1,\ldots,t_n)] \qquad \text{by B.8 since } f(t_1,\ldots,t_n) \in T_\Pi$$

which concludes the proof.   □

Finally, we show that $\mathcal{M}$ is a model of $=_\mathsf{x}$ and E.

Lemma B.11 $\mathcal{M} \models E, =_\mathsf{x}$

Proof. Since $\mathcal{M}$ is a structure whose domain $\mathcal{R}$ is the domain of semantic values of X, and since the interpretation in $\mathcal{M}$ of every interpreted symbol f is precisely its interpretation in X, namely f^X, $\mathcal{M}$ is a model of $=_\mathsf{x}$.

Moreover, let $t = u$ be an equation in E. Since t and u are in T_Π, the preceding lemma tells us that $\mathcal{M}(t) \equiv \Delta_\omega[t]$ and $\mathcal{M}(u) \equiv \Delta_\omega[u]$. By proposition B.3, we know that since $t = u$ is in E, it has been treated at the end and $\Delta_\omega[t] \equiv \Delta_\omega[u]$. Thus, $\mathcal{M}(t) \equiv \mathcal{M}(u)$ for any equation $t = u$ in E, and $\mathcal{M} \models E$.   □

Theorem B.12 (Completeness) $E, =_X \models a = b \;\Rightarrow\; \Delta_\omega[a] \equiv \Delta_\omega[b]$.

Proof. By lemma B.11, $\mathcal{M}$ is a model of E and $=_\mathsf{x}$. Therefore, since $E, =_\mathsf{x} \models a = b$, it must be the case that $\mathcal{M}$ is also a model of $a = b$, in other words, that $\mathcal{M}(a) \equiv \mathcal{M}(b)$. Hence, by lemma B.10, $\Delta_\omega[a] \equiv \Delta_\omega[b]$.   □