Rewriting-based Quantifier-free Interpolation for a Theory of Arrays

Roberto Bruttomesso[1], Silvio Ghilardi[2], and Silvio Ranise[3]

1 **Università della Svizzera Italiana, Lugano, Switzerland**
2 **Università degli Studi di Milano, Milan, Italy**
3 **FBK (Fondazione Bruno Kessler), Trento, Italy**

Abstract

The use of interpolants in model checking is becoming an enabling technology to allow fast and robust verification of hardware and software. The application of encodings based on the theory of arrays, however, is limited by the impossibility of deriving quantifier-free interpolants in general. In this paper, we show that, with a minor extension to the theory of arrays, it is possible to obtain quantifier-free interpolants. We prove this by designing an interpolating procedure, based on solving equations between array updates. Rewriting techniques are used in the key steps of the solver and its proof of correctness. To the best of our knowledge, this is the first successful attempt of computing quantifier-free interpolants for a theory of arrays.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.171

Category Regular Research Paper

1 Introduction

After the seminal work of McMillan (see, e.g., [20]), Craig's interpolation [9] has become an important technique in verification. For example, the importance of computing *quantifier-free* interpolants to over-approximate the set of reachable states for model checking has been observed. Unfortunately, Craig's interpolation theorem does not guarantee that it is always possible to compute quantifier-free interpolants. Even worse, for certain first-order theories, it is known that quantifiers must occur in interpolants of quantifier-free formulae [15]. As a consequence, a lot of effort has been put in designing efficient procedures for the computation of quantifier-free interpolants for first-order theories which are relevant for verification (e.g., uninterpreted functions and fragments of Presburger arithmetics). Despite these efforts, so far, only the negative result in [15] is available for the computation of interpolants in the theory of arrays with extensionality, axiomatized by the following three sentences: $\forall y, i, e.rd(wr(y, i, e), i) = e$, $\forall y, i, j, e.i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j)$, and

$$\forall x, y.x \neq y \Rightarrow (\exists i.\ rd(x, i) \neq rd(y, i)),$$

where rd and wr are the usual operations for reading and updating arrays, respectively. This theory is important for both hardware and software verification, and a procedure for computing quantifier-free interpolants "*would extend the utility of interpolant extraction as a tool in the verifier's toolkit*" [20]. Indeed, the endeavour of designing such a procedure would be bound to fail (according to [15]) if we restrict ourselves to the original theory. To circumvent the problem, we replace the third axiom above with its Skolemization, i.e.,

$$\forall x, y.x \neq y \Rightarrow rd(x, \mathtt{diff}(x, y)) \neq rd(y, \mathtt{diff}(x, y))),$$

so that the Skolem function `diff` is supposed to return an index at which the elements stored in two distinct arrays are different. This variant of the theory of arrays admits quantifier-free interpolants for quantifier-free formulae. The main contribution of the paper is to prove

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 171–186
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this by designing an **algorithm for the generation of quantifier-free interpolants from finite sets** (intended conjunctively) **of literals in the theory of arrays with** `diff`. The algorithm uses as a sub-module a satisfiability procedure for sets of literals of the theory, based on a sequence of syntactic transformations organized in several groups. The most important group of such transformations is a Knuth-Bendix completion procedure (see, e.g., [2]) extended to solve an equation $a = wr(b, i, e)$ for b when this is required by the ordering defined on terms. The goal of these transformations is to produce a "modularized" constraint for which it is trivial to establish satisfiability. To compute interpolants, the satisfiability procedure is invoked on two mutually unsatisfiable sets A and B of literals. While running, the two instances of the procedure exchange literals on the common signature of A and B (similarly to the Nelson and Oppen combination method, see, e.g., [21]) and perform some additional operations. At the end of the computation, the execution trace is examined and the desired interpolant is built by applying simple rules manipulating Boolean combinations of literals in the common signature of A and B.

The paper is organized as follows. In §2, we recall some background notions and introduce the notation. In §3, we give the notion of modularized constraint and state its key properties. In §4, we describe the satisfiability solver for the theory of arrays with `diff` and extend it to produce interpolants in §5. Finally, we discuss the related work and conclude in §6. All proofs can be found in [5].

2 Background and Preliminaries

We assume the usual syntactic (e.g., signature, variable, term, atom, literal, formula, and sentence) and semantic (e.g., structure, truth, satisfiability, and validity) notions of first-order logic. The equality symbol "=" is included in all signatures considered below. For clarity, we shall use "≡" in the meta-theory to express the syntactic identity between two symbols or two strings of symbols.

A *theory* T is a pair (Σ, Ax_T), where Σ is a signature and Ax_T is a set of Σ-sentences, called the axioms of T (we shall sometimes write directly T for Ax_T). The Σ-structures in which all sentences from Ax_T are true are the *models* of T. A Σ-formula ϕ is *T-satisfiable* if there exists a model $\mathcal{M}$ of T such that ϕ is true in $\mathcal{M}$ under a suitable assignment $\mathtt{a}$ to the free variables of ϕ (in symbols, $(\mathcal{M}, \mathtt{a}) \models \phi$); it is *$T$-valid* (in symbols, $T \vdash \varphi$) if its negation is T-unsatisfiable or, equivalently, iff φ is provable from the axioms of T in a complete calculus for first-order logic. A formula φ_1 *T-entails* a formula φ_2 if $\varphi_1 \to \varphi_2$ is *T-valid*; the notation used for such T-entailment is $A \vdash_T B$ or simply $A \vdash B$, if T is clear from the context. The *satisfiability modulo the theory* T ($SMT(T)$) *problem* amounts to establishing the T-satisfiability of quantifier-free Σ-formulae.

Let T be a theory in a signature Σ; a *T-constraint* (or, simply, a constraint) A is a set of ground literals in a signature Σ' obtained from Σ by adding a set of free constants. Taking conjunction, we can see a finite constraint A as a single formula; thus, when we say that a constraint A is *T-satisfiable* (or just "satisfiable" if T is clear from the context), we mean that the associated formula (also called A) is satisfiable in a Σ'-structure which is a model of T. We have two notions of equivalence between constraints, which are summarized in the next definition:

▶ **Definition 2.1.** Let A and B be finite constraints (or, more generally, first order sentences) in an expanded signature. We say that A and B are *logically equivalent* (modulo T) iff $T \vdash A \leftrightarrow B$; on the other hand, we say that they are *$\exists$-equivalent* (modulo T) iff $T \vdash A^{\exists} \leftrightarrow B^{\exists}$, where $A^{\exists}$ (and similarly $B^{\exists}$) is the formula obtained from A by replacing free constants

with variables and then existentially quantifying them out.

Logical equivalence means that the constraints have the same semantic content (modulo T); $\exists$-equivalence is also useful because we are mainly interested in T-satisfiability of constraints and it is trivial to see that $\exists$-equivalence implies equi-satisfiability (again, modulo T). As an example, if we take a constraint A, we replace all occurrences of a certain term t in it by a fresh constant a and add the equality $a = t$, called the *(explicit) definition (of t)*, the constraint A' we obtain in this way is $\exists$-equivalent to A. As another example, suppose that $A \vdash_T a = t$, that a does not occur in t, and that A' is obtained from A by replacing a by t everywhere; then the following four constraints are $\exists$-equivalent

$$A, \quad A \cup \{a = t\}, \quad A' \cup \{a = t\}, \quad A'$$

(the first three are also pairwise logically equivalent). The above examples show how explicit definitions can be introduced and removed from constraints while preserving $\exists$-equivalence.

Theories of Arrays. In this paper, we consider a variant of a three-sorted theory of arrays defined as follows. The McCarthy *theory of arrays* $\mathcal{AX}$ [17] has three sorts `ARRAY`, `ELEM`, `INDEX` (called "array", "element", and "index" sort, respectively) and two function symbols rd and wr of appropriate arities; its axioms are:

$$\forall y, i, e. \quad rd(wr(y, i, e), i) = e \tag{1}$$

$$\forall y, i, j, e. \quad i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j). \tag{2}$$

The theory of *arrays with extensionality* $\mathcal{AX}_{\text{ext}}$ has the further axiom $\forall x, y. x \neq y \Rightarrow (\exists i.\ rd(x, i) \neq rd(y, i))$ (called the 'extensionality' axiom). To build the *theory of arrays with* `diff` $\mathcal{AX}_{\texttt{diff}}$, we need a further function symbol `diff` in the signature and we replace the extensionality axiom by its Skolemization

$$\forall x, y. \quad x \neq y \Rightarrow rd(x, \texttt{diff}(x, y)) \neq rd(y, \texttt{diff}(x, y)). \tag{3}$$

As it is evident from axiom (3), the new symbol `diff` is a binary function of sort `INDEX` taking two arguments of sort `ARRAY`: its semantics is a function producing an index where the input arguments differ (it has an arbitrary value in case the input arguments are equal).

We introduce here some notational conventions which are specific for constraints in our theory $\mathcal{AX}_{\texttt{diff}}$. We use $a, b, \ldots$ to denote free constants of sort `ARRAY`, $i, j, \ldots$ for free constants of sort `INDEX`, and $d, e, \ldots$ for free constants of sort `ELEM`; $\alpha, \beta, \ldots$ stand for free constants of any sort. Below, we shall introduce non-ground rewriting rules involving (universally quantified) variables of sort `ARRAY`: for these variables, we shall use the symbols $x, y, z, \ldots$. We make use of the following abbreviations.

- [Nested write terms] By $wr(a, I, E)$ we indicate a nested write on the array variable a, where indexes are represented by the free constants list $I \equiv i_1, \ldots, i_n$ and elements by the free constants list $E \equiv e_1, \ldots, e_n$; more precisely, $wr(a, I, E)$ abbreviates the term $wr(wr(\cdots wr(a, i_1, e_1) \cdots), i_n, e_n)$. Notice that, whenever the notation $wr(a, I, E)$ is used, the lists I and E must have the same length; for empty I, E, the term $wr(a, I, E)$ conventionally stands for a.
- [Multiple read literals] Let a be a constant of sort `ARRAY`, $I \equiv i_1, \ldots, i_n$ and $E \equiv e_1, \ldots, e_n$ be lists of free constants of sort `INDEX` and `ELEM`, respectively; $rd(a, I) = E$ abbreviates the formula $rd(a, i_1) = e_1 \wedge \cdots \wedge rd(a, i_n) = e_n$.
- [Multiple equalities] If $L \equiv \alpha_1, \ldots, \alpha_n$ and $L' \equiv \alpha'_1, \ldots, \alpha'_n$ are lists of constants of the same sort, by $L = L'$ we indicate the formula $\bigwedge_{i=1}^{n} \alpha_i = \alpha'_i$.

Refl	$wr(a,I,E) = a \leftrightarrow rd(a,I) = E$ *Proviso*: $Distinct(I)$
Symm	$(wr(a,I,E) = b \wedge rd(a,I) = D) \leftrightarrow (wr(b,I,D) = a \wedge rd(b,I) = E)$ *Proviso*: $Distinct(I)$
Trans	$(a = wr(b,I,E) \wedge b = wr(c,J,D)) \leftrightarrow (a = wr(c, J \cdot I, D \cdot E) \wedge b = wr(c,J,D))$
Confl	$b = wr(a, I \cdot J, E \cdot D) \wedge b = wr(a, I \cdot H, E' \cdot F) \leftrightarrow$ $\leftrightarrow (b = wr(a,I,E) \wedge E = E' \wedge rd(a,J) = D \wedge rd(a,H) = F)$ *Proviso*: $Distinct(I \cdot J \cdot H)$
Red	$(a = wr(b,I,E) \wedge rd(b,i_k) = e_k) \leftrightarrow (a = wr(b, I-k, E-k) \wedge rd(b,i_k) = e_k)$ *Proviso*: $Distinct(I)$

Legenda: a and b are constants of sort `ARRAY`; $I \equiv i_1, \ldots, i_n$, $J \equiv j_1, \ldots, j_m$ and $H \equiv h_1, \ldots, h_l$ are lists of constants of sort `INDEX`; $E \equiv e_1, \ldots, e_n$, $E' \equiv e'_1, \ldots, e'_n$, $D \equiv d_1, \ldots, d_m$, and $F \equiv f_1, \ldots, f_l$ are lists of constants of sort `ELEM`.

Figure 1 Key properties of write terms

- [Multiple distinctions] If $L \equiv \alpha_1, \ldots, \alpha_n$ is a list of constants of the same sort, by $Distinct(L)$ we abbreviate the formula $\bigwedge_{i \neq j} \alpha_i \neq \alpha_j$.
- [Juxtaposition and subtraction] If $L \equiv \alpha_1, \ldots, \alpha_n$ and $L' \equiv \alpha'_1, \ldots, \alpha'_m$ are lists of constants, by $L \cdot L'$ we indicate the list $\alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_m$; for $1 \leq k \leq n$, the list $L - k$ is the list $\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots, \alpha_n$.

Some key properties of equalities involving write terms are stated in the following lemma (see also Figure 1).

▶ **Lemma 2.2** (Key properties of write terms). *The formulae in Figure 1 are all $\mathcal{AX}_{\texttt{diff}}$-valid under the assumption that their provisoes - if any - hold (when we say that a formula ϕ is $\mathcal{AX}_{\texttt{diff}}$-valid under the proviso π, we just mean that $\pi \vdash_{\mathcal{AX}_{\texttt{diff}}} \phi$).*

A (ground) *flat* literal is a literal of the form $a = wr(b,I,E), rd(a,i) = e, \texttt{diff}(a,b) = i, \alpha = \beta, \alpha \neq \beta$. Notice that replacing a sub-term t with a fresh constant α in a constraint A and adding the corresponding defining equation $\alpha = t$ to A always produces an $\exists$-equivalent constraint; by repeatedly applying this method, one can show that every constraint is $\exists$-equivalent to a *flat* constraint, i.e., to one containing only flat literals. We split a flat constraint A into two parts, the *index* part A_I and the *main* part A_M: A_I contains the literals of the form $i = j, i \neq j, \texttt{diff}(a,b) = i$, whereas A_M contains the remaining literals, i.e., those of the form $a = wr(b,I,E), a \neq b, rd(a,i) = e, e = d, e \neq d$ (atoms $a = b$ are identified with literals $a = wr(b, \emptyset, \emptyset)$). We write $A =< A_I, A_M >$ to indicate the two parts of the constraint A.

3 Constraints combination

We shall need basic term rewriting system terminology and results: the reader is referred to [2] for the required background. In the main part of a constraint, positive literals will be treated as rewrite rules; to get a suitable orientation, we use a *lexicographic path ordering* with a total precedence $>$ such that $a > wr > rd > \texttt{diff} > i > e$, for all a, i, e of the corresponding sorts. This choice orients equalities $a = wr(b,I,E)$ *from left to right* when $a > b$; equalities like $a = wr(b,I,E)$ for $a < b$ or $a \equiv b$ will be called *badly orientable* equalities. Our plan to derive a quantifier-free interpolation procedure for $\mathcal{AX}_{\texttt{diff}}$ relies on

the notion of "modularized constraint": after introducing such constraints, we show that their satisfiability can be easily recognized (Lemma 3.2) and that they can be combined in a modular way (Proposition 3.3).

▶ **Definition 3.1.** A constraint $A =< A_I, A_M >$ is said to be *modularized* iff it is flat and the following conditions are satisfied (we let $\tilde{I}, \tilde{E}$ be the sets of free constants of sort `INDEX` and `ELEM` occurring in A):

(o) no positive index literal $i = j$ occurs in A_I;

(i) no negative array literal $a \neq b$ occurs in A_M;

(ii) A_M does not contain badly orientable equalities;

(iii) the rewriting system A_R given by the oriented positive literals of A_M joined with the rewriting rules

$$rd(wr(x,i,e),j) \to rd(x,j) \qquad \text{for } i,j \in \tilde{I},\ e \in \tilde{E},\ i \not\equiv j \tag{4}$$

$$rd(wr(x,i,e),i) \to e \qquad \text{for } i \in \tilde{I},\ e \in \tilde{E} \tag{5}$$

$$wr(wr(x,i,e),j,d) \to wr(wr(x,j,d),i,e) \qquad \text{for } i,j \in \tilde{I},\ e,d \in \tilde{E},\ i > j \tag{6}$$

$$wr(wr(x,i,e),i,d) \to wr(x,i,d). \qquad \text{for } i \in \tilde{I},\ e,d \in \tilde{E} \tag{7}$$

is confluent and ground irreducible;[1]

(iv) if $a = wr(b,I,E) \in A_M$ and i, e are in the same position in the lists I, E, respectively, then $rd(b,i) \not\downarrow_{A_R} e$ (we use $\downarrow_{A_R}$ for joinability of terms);

(v) $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$ and $a \downarrow_{A_R} a'$ and $b \downarrow_{A_R} b'$ imply $i \equiv i'$;

(vi) $\mathtt{diff}(a,b) = i \in A_I$ and $rd(a,i) \downarrow_{A_R} rd(b,i)$ imply $a \downarrow_{A_R} b$.

▶ Remark. Condition (o) means that the index constants occurring in a modularized constraint are implicitly assumed to denote distinct objects. This is confirmed also by the proof of Lemma 3.2 below: from which, it is evident that the addition of all the negative literals $i \neq j$ (for $i, j \in \tilde{I}$ with $i \not\equiv j$) does not compromise the satisfiability of a modularized constraint, precisely because such negative literals are implicitly part of the constraint.

In Condition (i), negative array literals $a \neq b$ are not allowed because they can be replaced by suitable literals involving fresh constants and the `diff` operation (see axiom (3)).

Rules (4) and (5) mentioned in condition (iii) reduce read-over-writes and rules (6) and (7) sort indexes in flat terms $wr(a,I,E)$ in ascending order. In addition, condition (iv) prevents further redundancies in our rules.

Conditions (v) and (vi) deal with `diff`: in particular, (v) says that `diff` is "well defined" and (vi) is a "conditional" translation of the contraposition of axiom (3).

▶ Remark. The non-ground rules from Definition 3.1(iii) form a convergent rewrite system (critical pairs are confluent): this can be checked manually (and can be confirmed also by tools like SPASS or MAUDE). Ground rules from A_R are of the form

$$a \to wr(b,I,E), \tag{8}$$

$$rd(a,i) \to e, \tag{9}$$

$$e \to d. \tag{10}$$

Only rules of the form (10) can overlap with the non-ground rules (4)-(7), but the resulting critical pairs are trivially confluent. Thus, in order to check confluence of A_M, *only overlaps*

[1] The latter means that no rule can be used to reduce the left-hand or the right-hand side of another ground rule. Notice that ground rules from A_R are precisely the rules obtained by orienting an equality from A_M (rules (4)-(7) are not ground as they contain one *variable*, namely the array variable x).

between ground rules (8)-(10) *need to be considered* (this is the main advantage of our choice to orient equalities $a = wr(b, I, E)$ from left to right instead of right to left).

▶ **Lemma 3.2.** *A modularized constraint A is $\mathcal{AX}_{\mathtt{diff}}$-satisfiable iff for no negative element equality $e \neq d$ from A_M, we have that $e \downarrow_{A_R} d$.*

Let A, B be two constraints in the signatures Σ^A, Σ^B obtained from the signature Σ by adding some free constants and let $\Sigma^C := \Sigma^A \cap \Sigma^B$. Given a term, a literal or a formula φ we call it:

- *AB-common* iff it is defined over Σ^C;
- *A-local* (resp. *B-local*) if it is defined over Σ^A (resp. Σ^B);
- *A-strict* (resp. *B-strict*) iff it is A-local (resp. B-local) but not AB-common;
- *AB-mixed* if it contains symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$;
- *AB-pure* if it does not contain symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$.

(Notice that, sometimes in the literature about interpolation, "A-local" and "B-local" are used to denote what we call here "A-strict" and "B-strict"). The following modularity result is crucial for establishing interpolation in $\mathcal{AX}_{\mathtt{diff}}$:

▶ **Proposition 3.3.** *Let $A = \langle A_I, A_M \rangle$ and $B = \langle B_I, B_M \rangle$ be constraints in expanded signatures Σ^A, Σ^B as above (here Σ is the signature of $\mathcal{AX}_{\mathtt{diff}}$); let A, B be both consistent and modularized. Then $A \cup B$ is consistent and modularized, in case all the following conditions hold:*

(O) *an AB-common literal belongs to A iff it belongs to B;*

(I) *every rewrite rule in $A_M \cup B_M$ whose left-hand side is AB-common has also an AB-common right-hand side;*

(II) *if a, b are both AB-common and $\mathtt{diff}(a, b) = i \in A_I \cup B_I$, then i is AB-common too;*

(III) *if a rewrite rule of the kind $a \to wr(c, I, E)$ is in $A_M \cup B_M$ and the term $wr(c, I, E)$ is AB-common, so is the constant a.*

4 A Solver for Arrays with `diff`

In this section we present a solver for the theory $\mathcal{AX}_{\mathtt{diff}}$. The idea underlying our algorithm is to separate the "index" part (to be treated by guessing) of a constraint from the "array" and "elem" parts (to be treated with rewriting techniques). The problem is how, given a *finite* constraint A, to determine whether it is satisfiable or not by transforming it into a modularized $\exists$-equivalent constraint.

4.1 Preprocessing

In order to establish the satisfiability of a constraint A, we first need a pre-processing phase, consisting of the following sequential steps:

Step 1 Flatten A, by replacing sub-terms with fresh constants and by adding the related defining equalities.

Step 2 Replace array inequalities $a \neq b$ by the following literals (i, e, d are fresh)

$$\mathtt{diff}(a, b) = i, \quad rd(b, i) = e, \quad rd(a, i) = d, \quad d \neq e.$$

Step 3 Guess a partition of index constants, i.e., for any pair of indexes i, j add either $i = j$ or $i \neq j$ (but not both of them); then remove the positive literals $i = j$ by replacing i by j everywhere (if $i > j$ according to the symbol precedence, otherwise replace j by

i); if an inconsistent literal $i \neq i$ is produced, try with another guess (and if all guesses fail, report `unsat`).

Step 4 For all a, i such that $rd(a, i) = e$ does not occur in the constraint, add such a literal $rd(a, i) = e$ with fresh e.

At the end of the preprocessing phase, we get a finite set of flat constraints; *the disjunction of these constraints is $\exists$-equivalent to the original constraint.* For each of these constraints, go to the completion phase: *if the transformations below can be exhaustively applied (without failure) to at least one of the constraints, report* `sat`, *otherwise report* `unsat`.

The reason for inserting Step 4 above is just to simplify Orientation and Gaussian completion below. Notice that, even if rules $rd(a, i) \to e$ can be removed during completion, the following **invariant** is maintained: *terms $rd(a, i)$ always reduce to constants of sort* `ELEM`.

4.2 Completion

The completion phase consists in various transformations that should be non-deterministically executed until no rule or a failure instruction applies. For clarity, we divide the transformations into five groups.

(I) Orientation. This group contains a single instruction: get rid of badly orientable equalities, by using the equivalences **Refl**exivity and **Symm**etry of Figure 1; a badly orientable equality $a = wr(b, I, E)$ (with $a < b$) is replaced by an equality of the kind $b = wr(a, I, D)$ and by the equalities $rd(a, I) = E$ (all "read literals" required by the left-hand side of **Symm** comes from the above invariant). A badly orientable equality $a = wr(a, I, E)$ is removed and replaced by read literals only (or by nothing if I, E are empty).

(II) Gaussian completion. We now take care of the confluence of A_R (i.e., point (iii) of Definition 3.1). To this end, we consider all the critical pairs that may arise among our rewriting rules (8)-(10) (recall that, by Remark 3, there is no need to examine overlaps involving the non ground rules (4)-(7)). To treat the relevant critical pairs, we combine standard Knuth-Bendix completion for congruence closure with a specific method ("Gaussian completion") based on equivalences **Symm**etry, **Trans**itivity and **Confl**ict of Figure 1.[2] The critical pairs are listed below. Two preliminary observations are in order. First, we normalize a critical pair by using $\to_*$ before recovering convergence by adding a suitably oriented equality and removing the parent equalities (the symbol $\to_*$ denotes the reflexive and transitive closure of the rewrite relation $\to$ induced by the rewrite rules $A_R \cup \{(4)-(7)\}$). Second, the provisoes of all the equivalences in Figure 1 used below (i.e., **Symm**, **Trans**, and **Confl**) are satisfied because of the pre-processing Step 3 above.

(C1) $wr(b_1, I_1, E_1) \;{}_*\!\leftarrow wr(b'_1, I'_1, E'_1) \leftarrow a \to wr(b'_2, I'_2, E'_2) \to_* wr(b_2, I_2, E_2)$

with $b_1 > b_2$. We proceed in two steps. First, we use **Symm** (from right to left) to replace the parent rule $a \to wr(b'_1, I'_1, E'_1)$ with

$$wr(a, I_1, F) = b_1 \wedge rd(a, I_1) = E_1$$

for a suitable list F of constants of sort `ELEM` (notice that the equalities $rd(b_1, I_1) = F$, which are required to apply **Symm**, are already available because terms of the form $rd(b_1, j)$ for j in I_1 always reduce to constants of sort `ELEM` by the invariant resulting from the application of Step 4 in the pre-processing phase). Then, we apply **Trans** to

[2] The name "Gaussian" is due to the analogy with Gaussian elimination in Linear Arithmetic (see [1,4] for a generalization to the first-order context).

the previously derived equality $b_1 = wr(a, I_1, F)$ and to the normalized second equality of the critical pair (i.e., $a = wr(b_2, I_2, E_2)$) and we derive

$$b_1 = wr(b_2, I_2 \cdot I_1, E_2 \cdot F) \wedge a = wr(b_2, I_2, E_2). \tag{11}$$

Hence, we are entitled to replace $b_1 = wr(a, I_1, F)$ with the rule $b_1 \to wr(b_2, J, D)$, where J and D are lists obtained by normalizing the right-hand-side of the first equality of (11) with respect to the non-ground rules (6) and (7). To summarize: the parent rules are removed and replaced by the rules

$$b_1 \to wr(b_2, J, D), \quad a \to wr(b_2, I_2, E_2)$$

and a bunch of new equalities of the form $rd(a, i) = e$, giving rise, in turn, to rules of the form $rd(b_2, i) \to e$ or to rewrite rules of the form (10) after normalization of their left members.

(C2) $\boxed{wr(b, I_1, E_1) \ {}_*\!\leftarrow wr(b'_1, I'_1, E'_1) \leftarrow a \to wr(b'_2, I'_2, E'_2) \to_* wr(b, I_2, E_2)}$

Since identities like $wr(c, H, G) = wr(c, \pi(H), \pi(G))$ are $\mathcal{AX}_{\texttt{diff}}$-valid for every permutation π (under the proviso $Distinct(H)$), it is harmless to suppose that the set of index variables $I := I_1 \cap I_2$ coincides with the common prefix of the lists I_1 and I_2; hence we have $I_1 \equiv I \cdot J$ and $I_2 \equiv I \cdot H$ for suitable disjoint lists J and H. Then, let E and E' be the prefixes of E_1 and E_2, respectively, of length equal to that of I; and let $E_1 \equiv E \cdot D$ and $E_2 \equiv E' \cdot F$ for suitable lists D and F. At this point, we can apply **Confl** to replace both parent rules forming the critical pair with

$$a = wr(b, I, E) \wedge E = E' \wedge rd(b, J) = D \wedge rd(b, H) = F,$$

where the first equality is oriented from left to right (i.e., $a \to wr(b, I, E)$).

(III) Knuth-Bendix completion. The remaining critical pairs are treated by standard completion methods for congruence closure.

(C3) $\boxed{d \ {}_*\!\leftarrow rd(wr(b, I, E), i) \leftarrow rd(a, i) \to e' \to_* e}$

Remove the parent rule $rd(a, i) \to e'$ and, depending on whether $d > e, e > d$, or $d \equiv e$, add the rule $d \to e$, $e \to d$, or do nothing. (Notice that terms of the form $rd(b, j)$ are always reducible because of the invariant of Step 4 in the pre-processing phase; hence, $rd(wr(b, I, E), i)$ always reduces to some constant of sort `ELEM`.)

(C4) $\boxed{e \ {}_*\!\leftarrow e' \leftarrow rd(a, i) \to d' \to_* d}$

Orient the critical pair (if e and d are not identical), add it as a new rule and remove one parent rule.

(C5) $\boxed{d \ {}_*\!\leftarrow d' \leftarrow e \to d'_1 \to_* d_1}$

Orient the critical pair (if d and d_1 are not identical), add it as a new rule and remove one parent rule.

(IV) Reduction. The instructions in this group simplify the current rewrite rules.

(R1) If the right-hand side of a current ground rewrite rule can be reduced, reduce it as much as possible, remove the old rule, and replace it with the newly obtained reduced rule. Identical equations like $t = t$ are also removed.

(R2) For every rule $a \to wr(b, I, E) \in A_M$, exhaustively apply **Red**uction in Figure 1 from left to right (this amounts to do the following: if there are i, e in the same position k in the lists I, E such that $rd(b, i) \downarrow_{A_R} e$, replace $a = wr(b, I, E)$ with $a = wr(b, I{-}k, E{-}k)$).

(R3) If $\texttt{diff}(a, b) = i \in A_I$, $rd(a, i) \downarrow_{A_R} rd(b, i)$ and $a > b$, add the rule $a \to b$; replace also $\texttt{diff}(a, b) = i$ by $\texttt{diff}(b, b) = i$ (this is needed for termination, it prevents the rule for being indefinitely applied).

(V) Failure. The instructions in this group aim at detecting inconsistency.

(U1) If for some negative literal $e \neq d \in A_M$ we have $e \downarrow_{A_R} d$, report `failure` and backtrack to Step 3 of the pre-processing phase.

(U2) If $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$ and $a \downarrow_{A_R} a'$ and $b \downarrow_{A_R} b'$ for $i \not\equiv i'$, report `failure` and backtrack to Step 3 of the pre-processing phase.

Notice that the instructions in the last two groups may require a confluence test $\alpha \downarrow_{A_R} \beta$ that can be effectively performed in case the instructions from groups (II)-(III) have been exhaustively applied, because then all critical pairs have been examined and the rewrite system A_R is confluent. If this is not the case, one may pragmatically compute and compare any normal form of α and β, keeping in mind that the test has to be repeated when all completion instructions (II)-(III) have been exhaustively applied.

▶ **Theorem 4.1.** *The above procedure decides constraint satisfiability in* $\mathcal{AX}_{\mathtt{diff}}$.

5 The Interpolation Algorithm

In the literature one can roughly distinguish two approaches to the problem of computing interpolants. In the former (see e.g. [3, 19]), an interpolating calculus is obtained from a standard calculus by adding decorations so as to enable the recursive construction of an interpolating formula from a proof; in the latter (see, e.g., [7, 11, 23]), the focus is on how to extend an available decision procedure to return interpolants. Our methodology is similar to the second approach, since we add the capability of computing interpolants to the satisfiability procedure in Section 4. However, we do this by designing a flexible and abstract framework, relying on the identification of *basic operations* that can be performed independently from the method used by the underlying satisfiability procedure to derive a refutation.

5.1 Interpolating Metarules

Let now A, B be constraints in signatures Σ^A, Σ^B expanded with free constants and $\Sigma^C := \Sigma^A \cap \Sigma^B$; we shall refer to the definitions of AB-common, A-local, B-local, A-strict, B-strict, AB-mixed, AB-pure terms, literals and formulae given in Section 3. Our goal is to produce, in case $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, a ground AB-common sentence ϕ such that $A \vdash_{\mathcal{AX}_{\mathtt{diff}}} \phi$ and $\phi \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable.

Let us examine some of the transformations to be applied to A, B. Suppose for instance that the literal ψ is AB-common and such that $A \vdash_{\mathcal{AX}_{\mathtt{diff}}} \psi$; then we can transform B into $B' := B \cup \{\psi\}$. Suppose now that we got an interpolant ϕ for the pair A, B': clearly, we can derive an interpolant for the original pair A, B by taking $\phi \wedge \psi$. The idea is to collect some useful transformations of this kind. Notice that these transformations can also modify the signatures Σ^A, Σ^B. For instance, suppose that t is an AB-common term and that c is a fresh constant: then we can put $A' := A \cup \{c = t\}$, $B' := B \cup \{c = t\}$: in fact, if ϕ is an interpolant for A', B', then $\phi(t/c)$ is an interpolant for A, B.[3] The transformations we need are called *metarules* and are listed in Table 1 below (in the Table and more generally in this Subsection, we use the notation $\phi \vdash \psi$ for $\phi \vdash_{\mathcal{AX}_{\mathtt{diff}}} \psi$).

[3] Notice that the fresh constant c is now a shared symbol, because Σ^A is enlarged to $\Sigma^A \cup \{c\}$, Σ^B is enlarged to $\Sigma^B \cup \{c\}$ and hence $(\Sigma^A \cup \{c\}) \cap (\Sigma^B \cup \{c\}) = \Sigma^C \cup \{c\}$.

An *interpolating metarules refutation* for A, B is a labeled tree having the following properties: (i) nodes are labeled by pairs of finite sets of constraints; (ii) the root is labeled by A, B; (iii) the leaves are labeled by a pair A, B such that $\bot \in A \cup B$; (iv) each non-leaf node is the conclusion of a rule from Table 1 and its successors are the premises of that rule. The crucial properties of the metarules are summarized in the following two Propositions.

▶ **Proposition 5.1.** *The unary metarules* $\frac{A \mid B}{A' \mid B'}$ *from Table 1 have the property that* $A \wedge B$ *is* $\exists$*-equivalent to* $A' \wedge B'$*; similarly, the n-ary metarules* $\frac{A_1 \mid B_1 \quad \cdots \quad A_n \mid B_n}{A \mid B}$ *from Table 1 have the property that* $A \wedge B$ *is* $\exists$*-equivalent to* $\bigvee_{k=1}^{n}(A_k \wedge B_k)$.

▶ **Proposition 5.2.** *If there exists an interpolating metarules refutation for* A, B *then there is a quantifier-free interpolant for* A, B *(namely there exists a quantifier-free* AB*-common sentence* ϕ *such that* $A \vdash \phi$ *and* $B \wedge \phi \vdash \bot$*). The interpolant* ϕ *is recursively computed applying the relevant interpolating instructions from Table 1.*

5.2 The Interpolation Solver

The metarules are complete, i.e. if $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, then (since we shall prove that an interpolant exists) a single application of (Propagate1) and (Close2) gives an interpolating metarules refutation. This observation shows that metarules are by no means better than the brute force enumeration of formulae to find interpolants. However, metarules are useful to design an algorithm manipulating pairs of constraints based on transformation instructions. In fact, each of the transformation instructions can be *justified* by a metarule (or by a sequence of metarules): in this way, if our instructions form a complete and terminating algorithm, we can use Proposition 5.2 to get the desired interpolants. The main advantage of using metarules as justifications is that we just need to take care of the completeness and termination of the algorithm, and not about interpolants anymore. Here "completeness" means that our transformations should be able to bring a pair (A, B) of constraints into a pair (A', B') that either matches the requirements of Proposition 3.3 or is explicitly inconsistent, in the sense that $\bot \in A' \cup B'$. The latter is obviously the case whenever the original pair (A, B) is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable and it is precisely the case leading to an interpolating metarules refutation.

The basic idea is that of invoking the algorithm of Section 4 on A and B separately and to propagate equalities involving AB-common terms. We shall assume *an ordering precedence making* AB*-common constants smaller than* A*-strict or* B*-strict constants of the same sort.* However, this is not sufficient to prevent the algorithm of Section 4 from generating literals and rules violating one or more of the hypotheses of Proposition 3.3: this is why the extra correcting instructions of group (γ) below are needed. Our interpolating algorithm has a pre-processing and a completion phase, like the algorithm from Section 4.

Pre-processing. In this phase the four Steps of Section 4.1 are performed both on A and on B; to justify these steps we need metarules (Define0,1,2), (Redplus1,2), (Redminus1,2), (Disjunction1,2), (ConstElim0,1,2), and (Propagate1,2) - the latter because if i, j are AB-common, the guessing of $i = j$ versus $i \neq j$ in Step 3 can be done, say, in the A-component and then propagated to the B-component. At the end of the preprocessing phase, the following properties (to be maintained as invariants afterwards) hold:

(i1) A (resp. B) contains $i \neq j$ for all A-local (resp. B-local) constants i, j of sort `INDEX` occurring in A (resp. in B);

(i2) if a, i occur in A (resp. in B), then $rd(a, i)$ reduces to an A-local (resp. B-local) constant of sort `ELEM`.

Close1	Close2	Propagate1	Propagate2
$\dfrac{}{A \mid B}$	$\dfrac{}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$
Prv.: A is unsat. *Int.*: $\phi' \equiv \bot$.	*Prv.*: B is unsat. *Int.*: $\phi' \equiv \top$.	*Prv.*: $A \vdash \psi$ and ψ is AB-common. *Int.*: $\phi' \equiv \phi \wedge \psi$.	*Prv.*: $B \vdash \psi$ and ψ is AB-common. *Int.*: $\phi' \equiv \psi \rightarrow \phi$.

Define0	Define1	Define2
$\dfrac{A \cup \{a = t\} \mid B \cup \{a = t\}}{A \mid B}$	$\dfrac{A \cup \{a = t\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{a = t\}}{A \mid B}$
Prv.: t is AB-common, a fresh. *Int.*: $\phi' \equiv \phi(t/a)$.	*Prv.*: t is A-local and a is fresh. *Int.*: $\phi' \equiv \phi$.	*Prv.*: t is B-local and a is fresh. *Int.*: $\phi' \equiv \phi$.

Disjunction1	Disjunction2
$\dfrac{\cdots \quad A \cup \{\psi_k\} \mid B \quad \cdots}{A \mid B}$	$\dfrac{\cdots \quad A \mid B \cup \{\psi_k\} \quad \cdots}{A \mid B}$
Prv.: $\bigvee_{k=1}^{n} \psi_k$ is A-local and $A \vdash \bigvee_{k=1}^{n} \psi_k$. *Int.*: $\phi' \equiv \bigvee_{k=1}^{n} \phi_k$.	*Prv.*: $\bigvee_{k=1}^{n} \psi_k$ is B-local and $B \vdash \bigvee_{k=1}^{n} \psi_k$. *Int.*: $\phi' \equiv \bigwedge_{k=1}^{n} \phi_k$.

Redplus1	Redplus2	Redminus1	Redminus2
$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \mid B}{A \cup \{\psi\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{\psi\}}$
Prv.: $A \vdash \psi$ and ψ is A-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $B \vdash \psi$ and ψ is B-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $A \vdash \psi$ and ψ is A-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $B \vdash \psi$ and ψ is B-local. *Int.*: $\phi' \equiv \phi$.

ConstElim1	ConstElim2	ConstElim0
$\dfrac{A \mid B}{A \cup \{a = t\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{b = t\}}$	$\dfrac{A \mid B}{A \cup \{c = t\} \mid B \cup \{c = t\}}$
Prv.: a is A-strict and does not occur in A, t. *Int.*: $\phi' \equiv \phi$.	*Prv.*: b is B-strict and does not occur in B, t. *Int.*: $\phi' \equiv \phi$.	*Prv.*: c, t are AB-common, c does not occur in A, B, t. *Int.*: $\phi' \equiv \phi$.

Table 1 Interpolating Metarules: each rule has a proviso *Prv.* and an instruction *Int.* for recursively computing the new interpolant ϕ' from the old one(s) $\phi, \phi_1, \ldots, \phi_k$.

Completion. Some groups of instructions to be executed non-deterministically constitute the completion phase. There is however an important difference here with respect to the completion phase of Section 4.2: it may happen that we need some *guessing* also inside the completion phase (only the instructions from group (γ) below may need such guessings). Each instruction can be easily justified by suitable metarules (we omit the details for lack of space). The groups of instructions are the following:

(α) Apply to A or to B any instruction from the completion phase of Section 4.2.

(β) If there is an AB-common literal that belongs to A but not to B (or vice versa), copy it in B (resp. in A).

(γ) Replace *undesired literals*, i.e., those violating conditions (I)-(II)-(III) from Proposition 3.3.

To avoid trivial infinite loops with the (β) instructions, rules in (α) deleting an AB-common literal should be performed *simultaneously* in the A- and in the B-components (it can be easily checked [5] that this is always possible, for instance if rules in (β) and (γ) are given higher priority). Instructions (γ) need to be described in more details. Preliminarily, we introduce a technique that we call *Term Sharing*. Suppose that the A-component contains a literal $\alpha = t$, where the term t is AB-common but the free constant α is only A-local. Then it is possible to "make α AB-common" in the following way. First, introduce a fresh AB-common constant α' with the explicit definition $\alpha' = t$ (to be inserted both in A and in B, as justified by metarule (Define0)); then replace the literal $\alpha = t$ by $\alpha = \alpha'$ and replace α by α' everywhere else in A; finally, delete $\alpha = \alpha'$ too. The result is a pair (A, B) where basically nothing has changed but α has been renamed to an AB-common constant α'. Notice that the above transformations can be justified by metarules (Define0), (Redplus1), (Redminus1), (ConstElim1). We are now ready to explain instructions (γ) in details. First, consider undesired literals corresponding to the rewrite rules of the form

$$rd(c, i) \to d \tag{12}$$

in which the left-hand side is AB-common and the right-hand side is, say, A-strict. If we apply Term Sharing, we can solve the problem by renaming d to an AB-common fresh constant d'. We can apply a similar procedure to the rewrite rules

$$a \to wr(c, I, E) \tag{13}$$

in case the right-hand side is AB-common and the left-hand side is not; when we rename a to some fresh AB-common constant c', we must arrange the precedence so that $c' > c$ to orient the renamed literal as $c' \to wr(c, I, E)$. Then, consider the literals of the form

$$\texttt{diff}(a, b) = k \tag{14}$$

in which the left-hand side is AB-common and the right-hand side is, say, A-strict. Again, we can rename k to some AB-common constant k' by Term Sharing. Notice that k' is AB-common, whereas k was only A-local: this implies that we might need to perform some guessing to maintain the invariant (i1). Basically, we need to repeat Step 3 from Section 4.1 till invariant (i1) is restored (k' must be compared for equality with the other B-local constants of sort `INDEX`). The last undesired literals to take care of are the rules of the form[4]

$$c \to wr(c', I, E) \tag{15}$$

having an AB-common left-hand side but, say, only an A-local right-hand side. Notice that from the fact that c is AB-common, it follows (by our choice of the precedence) that c' is AB-common too. We can freely suppose that I and E are split into sublists I_1, I_2 and E_1, E_2, respectively, such that $I \equiv I_1 \cdot I_2$ and $E \equiv E_1 \cdot E_2$, where I_1, E_1 are AB-common, $I_2 \equiv i_1, \ldots, i_n$, $E_2 \equiv e_1, \ldots, e_n$ and for each $k = 1, \ldots, n$ at least one from i_k, e_k is not AB-common. This n (measuring essentially the number of non AB-common symbols in (15)) is called the *degree* of the undesired literal (15): in the following, we shall see how to eliminate (15) or to replace it with a smaller degree literal. We first make a guess (see metarule (Disjunction1)) about the truth value of the literal $c = wr(c', I_1, E_1)$. In the first

[4]Literals $d = e$ are automatically oriented in the right way by our choice of the precedence.

case, we add the positive literal to the current constraint; as a consequence, we get that the literal (15) is equivalent to $c = wr(c, I_2, E_2)$ and also to $rd(c, I_2) = E_2$ (see **Red** in Figure 1). In conclusion, in this case, the literal (15) is replaced by the AB-common rewrite rule $c \to wr(c', I_1, E_1)$ and by the literals $rd(c, I_2) = E_2$. In the second case, we guess that the negative literal $c \neq wr(c', I_1, E_1)$ holds; we introduce a fresh AB-common constant c'' together with the defining AB-common literal[5]

$$c'' \to wr(c', I_1, E_1) \tag{16}$$

(see metarule (Define0)). The literal (15) is replaced by the literal

$$c \to wr(c'', I_2, E_2). \tag{17}$$

We show how to make the degree of (17) smaller than n. In addition, we eliminate the negative literal $c \neq c''$ coming from our guessing (notice that, according to (16), c'' renames $wr(c', I_1, E_1)$). This is done as follows: we introduce fresh AB-common constants i, d, d'' together with the AB-common defining literals

$$\mathtt{diff}(c, c'') = i, \quad rd(c, i) \to d, \quad rd(c'', i) \to d'' \tag{18}$$

(see metarule (Define0)). Now it is possible to replace $c \neq c''$ by the literal $d \neq d''$ (see axiom (3)). Under the assumption $Distinct(I_2)$, the following statement is $\mathcal{AX}_{\mathtt{diff}}$ valid:

$$c = wr(c'', I_2, E_2) \wedge rd(c'', i) = d'' \wedge rd(c, i) = d \wedge d \neq d'' \to \bigvee_{k=1}^{n} (i = i_k \wedge d = e_k).$$

Thus, we get n alternatives (see metarule (Disjunction1)). In the k-th alternative, we can remove the constants i_k, e_k from the constraint, by replacing them with the AB-common terms i, d respectively (see metarules (Redplus1), (Redplus2), (Redminus1), (Redminus2),(ConstElim1),(ConstElim0)); notice that it might be necessary to complete the index partition. In this way, the degree of (17) is now smaller than n.

In conclusion, if we apply exhaustively Pre-Processing and Completion instructions above, starting from an initial pair of constraints (A, B), we can produce a tree, whose nodes are labelled by pairs of constraints (the successor nodes are labelled by pairs of constraints that are obtained by applying an instruction). We call such a tree an *interpolating tree* for (A, B). The following result shows that we obtained an interpolation algorithm for $\mathcal{AX}_{\mathtt{diff}}$:

▶ **Theorem 5.3.** *Any interpolation tree for (A, B) is finite; moreover, it is an interpolating metarules refutation (from which an interpolant can be recursively computed according to Proposition 5.2) precisely iff $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable.*

From the above Theorem it immediately follows that:

▶ **Theorem 5.4.** *The theory $\mathcal{AX}_{\mathtt{diff}}$ admits quantifier-free interpolants (i.e., for every quantifier free formulae ϕ, ψ such that $\psi \wedge \phi$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, there exists a quantifier free formula θ such that: (i) $\psi \vdash_{\mathcal{AX}_{\mathtt{diff}}} \theta$; (ii) $\theta \wedge \phi$ is not $\mathcal{AX}_{\mathtt{diff}}$-satisfiable: (iii) only variables occurring both in ψ and in ϕ occur in θ).*

In [5], we also give a direct (although non-constructive) proof of this theorem by using the model-theoretic notion of amalgamation.

[5] We put $c > c'' > c'$ in the precedence. Notice that invariant (i2) is maintained, because all terms $rd(c'', h)$ normalize to an element constant. In case I_1 is empty, one can directly take c' as c''.

5.3 An Example

To illustrate our method, we describe the computation of an interpolant for the mutually unsatisfiable sets $A \equiv \{a = wr(b,i,d)\}, B \equiv \{rd(a,j) \neq rd(b,j), rd(a,k) \neq rd(b,k), j \neq k\}$. Notice that i, d are A-strict constants, j, k are B-strict constants, and a, b are AB-common constants with precedence $a > b$. We first apply Pre-Processing instructions to obtain $A \equiv \{a = wr(b,i,d), rd(a,i) = e_5, rd(b,i) = e_6\}, B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k\}$. Since $a = wr(b,i,d)$ is an undesired literal of the kind (15), we generate the two subproblems $\Pi_1 \equiv (A \cup \{rd(b,i) = d, a = b\}, B)$ and $\Pi_2 \equiv (A \cup \{a \neq b\}, B)$.[6]

Let us consider Π_1 first. Notice that $A \vdash a = b$, and $a = b$ is AB-common. Therefore we send $a = b$ to B, and we may derive the new equality $e_1 = e_2$ from the critical pair (C3) $e_1 \leftarrow rd(a,j) \rightarrow rd(b,j) \rightarrow e_2$, thus obtaining $A \equiv \{a = b, rd(b,i) = d, rd(a,i) = e_5, rd(b,i) = e_6\}$, $B \equiv \{rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, a = b, e_1 = e_2\}$. Now B is inconsistent. The interpolant for Π_1 can be computed with the *interpolating instructions* of the metarules (Close1,Propagate1,Redminus1,Redplus1) resulting in $\varphi_1 \equiv (\top \wedge a = b) \equiv a = b$.

Then, let us consider branch Π_2. Recall that this branch originates from the attempt of removing the undesired rule $a \rightarrow wr(b,i,d)$. We introduce the AB-common defining literals $\mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2$, and $f_1 \neq f_2$, in order to remove $a \neq b$ from A. These are immediately propagated to B: $A \equiv \{a = wr(b,i,d), rd(a,i) = e_5, rd(b,i) = e_6, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}, B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}$. Since $a = wr(b,i,d)$ contains only the index i, we do not have a real case split. Therefore we replace i with l, and d with f_1. At last, we propagate the AB-common literal $a = wr(b,l,f_1)$ to B. After all these steps we obtain: $A \equiv \{a = wr(b,l,f_1), rd(a,l) = e_5, rd(b,i) = e_6, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}$, $B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2, a = wr(b,l,f_1)\}$. Since we have one more AB-common index constant l, we complete the current index constant partition, namely $\{k\}$ and $\{j\}$: we have three alternatives, to let l stay alone in a new class, or to add l to one of the two existing classes. In the first alternative, because of the following critical pair (C3) $e_1 \leftarrow rd(a,j) \rightarrow rd(wr(b,l,f_1),j) \rightarrow e_2$, we add $e_1 = e_2$ to B, which becomes trivially unsatisfiable. The other two alternatives yield similar outcomes. For each subproblem the interpolant, reconstructed by reverse application of the interpolating instructions of (Define0) and (Propagate1), is $\varphi_2' \equiv \{(a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b))) \wedge rd(a, \mathtt{diff}(a,b)) \neq rd(b, \mathtt{diff}(a,b)))\}$. The interpolant φ_2 for the branch Π_2 has to be computed by combining with (Disjunction2) three copies of φ_2', and so $\varphi_2 \equiv \varphi_2'$.

The final interpolant is computed by combining the interpolants for Π_1 and Π_2 by means of (Disjunction1), yielding $\varphi \equiv \varphi_1 \vee \varphi_2 \equiv (a = b \vee (a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b))) \wedge rd(a, \mathtt{diff}(a,b)) \neq rd(b, \mathtt{diff}(a,b))))$, i.e. $a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b)))$.

6 Related work and Conclusions

There is a series of papers devoted to building satisfiability procedures for the theory of arrays with or without extensionality. The interested reader is pointed to, e.g., [10, 12] for

[6] Notice that this is precisely the case in which there is no need of an extra AB-common constant c''.

an overview. In the following, for lack of space, we discuss the papers more closely related to interpolation for the theory of arrays.

After McMillan's seminal work on interpolation for model checking [18,20], several papers appeared whose aim was to design techniques for the efficient computation of interpolants in first-order theories of interest for verification, mainly uninterpreted function symbols, fragments of Linear Arithmetic, or their combination. An interpolating theorem prover is described in [19], where a sequent-like calculus is used to derive interpolants from proofs in propositional logic, equality with uninterpreted functions, linear rational arithmetic, and their combinations. In [15], a method to compute interpolants in data structures theories, such as sets and arrays (with extensionality), by axiom instantiation and interpolant computation in the theory of uninterpreted functions is described. It is also shown that the theory of arrays with extensionality does not admit quantifier-free interpolation. The "split" prover in [13] applies a sequent calculus for the synthesis of interpolants along the lines of that in [19] and is tuned for predicate abstraction [22]. The "split" prover can handle a combination of theories among which also the theory of arrays without extensionality is considered. In [13], it is pointed out that the theory of arrays poses serious problems in deriving quantifier-free interpolants because it entails an infinite set of quantifier-free formulae, which is indeed problematic when interpolants are to be used for predicate abstraction. To overcome the problem, [13] suggests to constrain array valued terms to occur in equalities of the form $a = wr(a, I, E)$ in the notation of this paper. It is observed that this corresponds to the way in which arrays are used in imperative programs. Further limitations are imposed on the symbols in the equalities in order to obtain a complete predicate abstraction procedure. In [14], the method described in [13] is specialized to apply CEGAR techniques [8] for the verification of properties of programs manipulating arrays. The method of [13] is extended to cope with range predicates which allow one to describe unbounded array segments which permit to formalize typical programming idioms of arrays, yielding property-sensitive abstractions. In [16], a method to derive quantified invariants for programs manipulating arrays and integer variables is described. A resolution-based prover is used to handle an *ad hoc* axiomatization of arrays by using predicates. Neither McCarthy's theory of arrays nor one of its extensions are considered in [16]. The invariant synthesis method is based on the computation of interpolants derived from the proofs of the resolution-based prover and constraint solving techniques to handle the arithmetic part of the problem. The resulting interpolants may contain even alternation of quantifiers.

To the best of our knowledge, the interpolation procedure presented in this paper is the first to compute quantifier-free interpolants for a natural variant of the theory of arrays with extensionality. In fact, the variant is obtained by replacing the extensionality axiom with its Skolemization which should be sufficient when the procedure is used to detect unsatisfiability of formulae as it is the case in standard model checking methods for infinite state systems. Because our method is not based on a proof calculus, we can avoid the burden of generating a large proof before being able to extract interpolants. The implementation of our procedure is currently being developed in the SMT-solver OpenSMT [6] and preliminary experiments are encouraging. An extensive experimental evaluation is planned for the immediate future.
Acknowledgements. We wish to thank an anonymuous referee for many useful criticisms that helped improving the quality of the paper.

References

1 F. Baader, S. Ghilardi, and C. Tinelli. A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. *Inform. and Comput.*,

204(10):1413–1452, 2006.

2 F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, 1998.

3 A. Brillout, D. Kroening, P. Rümmer, and W. Thomas. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic . In *IJCAR*, 2010.

4 R. Bruttomesso. *Problemi di combinazione nella dimostrazione automatica e nella verifica del software*. Università degli Studi di Milano, 2004. Master Thesis.

5 R. Bruttomesso, S. Ghilardi, and S. Ranise. Rewriting-based Quantifier-free Interpolation for a Theory of Arrays. Technical Report RI 334-10, Dip. Scienze dell'Informazione, Univ. di Milano, 2010.

6 R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *TACAS*, pages 150–153, 2010.

7 A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolation Generation in Satisfiability Modulo Theories. *ACM Trans. Comput. Logic*, 12:1–54, 2010.

8 E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.

9 W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, pages 269–285, 1957.

10 L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *FMCAD*, pages 45–52, 2009.

11 A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground Interpolation for the Theory of Equality. In *TACAS*, pages 413–427, 2009.

12 S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Annals of Mathematics and Artificial Intelligence*, 50:231–254, 2007.

13 R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, pages 459–473, 2006.

14 R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *CAV*, pages 193–206, 2007.

15 D. Kapur, R. Majumdar, and C. Zarba. Interpolation for Data Structures. In *SIGSOFT'06/FSE-14*, pages 105–116, 2006.

16 L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *FASE*, pages 470–485, 2009.

17 J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.

18 K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.

19 K. L. McMillan. An Interpolating Theorem Prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

20 K. L. McMillan. Applications of Craig Interpolation to Model Checking. In *TACAS*, pages 1–12, 2005.

21 S. Ranise, C. Ringeissen, and D. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *ICTAC*, pages 372–386, 2004.

22 H. Saidi and S. Graf. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

23 G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *CADE*, pages 353–368, 2005.