Weakly Equivalent Arrays

Jürgen Christ and Jochen Hoenicke*

Department of Computer Science,
University of Freiburg
{christj,hoenicke}@informatik.uni-freiburg.de

Abstract

The (extensional) theory of arrays is widely used to model systems. Hence, efficient decision procedures are needed to model check such systems. Current decision procedures for the theory of arrays saturate the read-over-write and extensionality axioms originally proposed by McCarthy. Various filters are used to limit the number of axiom instantiations while preserving completeness. We present an algorithm that lazily instantiates lemmas based on *weak equivalence classes*. These lemmas are easier to interpolate as they only contain existing terms. We formally define weak equivalence and show correctness of the resulting decision procedure.

1 Introduction

Arrays are widely used to model parts of systems. In software model checking, for example, the heap of a program can be modelled by an array that represents the main memory. A software model checker using such a model can check for illegal accesses to memory or even memory leaks. While checking for illegal accesses can be done using only the axioms proposed by McCarthy, leak checking typically is done using extensionality. In this setting, extensionality is used to ensure that the memory after executing a program does not contain more allocated memory cells than it contained at the beginning of the program.

The theory of arrays was initially proposed by McCarthy [7]. It specifies two operations: (1) The store operation $a\langle i \lhd v\rangle$ creates a new array that stores at every index different from i the same value as array a and the value v at index i. (2) The select operation $a[i]$ retrieves the value of array a at position i. The theory is parametric in the index and element theories.

The store operation only modifies an array at one index. The values stored at other indices are not affected by this operation. Hence, the resulting array and the array used in the store operation are *weakly equal* in the sense that they differ only at finitely many indices. Current decision procedures do not fully exploit such dependencies between arrays. Instead, they use a series of instantiations of the axiom proposed by McCarthy to derive weak equivalences.

In this paper we present a new algorithm to decide the quantifier-free fragment of the theory of arrays. The decision procedure is based on the notion of *weak equivalence*, a property that combines equivalence reasoning with array dependencies. The new algorithm only produces a few new terms not present in the input formula during preprocessing. This is possible since the decision procedure does not instantiate the axiom proposed by McCarthy, but axioms derived from them.

[*] This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS)

Related Work Since the proposal of the theory of arrays by McCarthy [7] several decision procedures have been proposed. We can identify two basic branches: *rewrite-based* and *instantiation-based* techniques.

Armando et al. [1] used rewriting techniques to solve the theory of arrays. They showed how to construct simplification orderings to achieve completeness. The benchmarks used in this paper test specific properties of the array operators like commutativity of stores if the indices differ. While these benchmarks require a lot of instantiations of McCarthy's axioms, they are easy for the decision procedure presented in this paper since the properties tested by these benchmarks are properties satisfied by the weak equivalence relation presented in this paper.

Bruttomesso et al. [4] present a rewrite based decision procedure to reason about arrays. This approach exploits some key properties of the store operation that are also captured by the weak equivalence relation described in this paper. Contrary to our method, the rewrite based approach is not designed for Nelson–Oppen style theory combination and thus not easily integratable into an existing SMT solver. They extended the solver into an interpolating solver for computing quantifier-free interpolants. In contrast to our method their solver depends on the partitioning of the interpolation problem. We create a SMT proof without any knowledge of the partitioning and can use proof tree preserving interpolation [6], which only requires a procedure to interpolate the lemmas.

A decision procedure for the theory of arrays based on instantiating McCarthy's axioms is given by de Moura et al. [8]. The decision procedure saturates several rules that instantiate array axioms under certain conditions. Several filters are proposed to minimise the number of instantiations.

Closest to our work is the decision procedure published by Brummayer et al. [3]. Their decision procedure produces lemmas that can be derived from the axioms for the theory of arrays proposed by McCarthy. They consider the theory of arrays with bitvector indices and prove soundness and completeness of their approach in this setting. In contrast to our method, they do not allow free function symbols (i. e., the combination of the theory of arrays with the theory of uninterpreted function symbols) since they only consider a limited form of extensionality where the extensionality axiom is only instantiated for arrays a and b if the formula contains the literal $a \neq b$. We do not have this limitation, but add some requirements on the index theory that prevent the procedure presented in this paper from using the theory of bitvectors as index theory.

2 Notation

A first order theory consists of a signature Σ and a set of models $\mathbb{M}$. We assume the equality symbol $=$ with its usual interpretation is part of any signature. Every model contains for every sort interpreted by this model a non-empty domain and a mapping from constant or function symbol into the corresponding domain. A theory $\mathcal{T}$ is *stably infinite* if and only if every satisfiable quantifier-free formula is satisfied in a model of $\mathcal{T}$ with an infinite universe.

The theory of arrays $\mathcal{T}_A$ is parameterised by an index theory $\mathcal{T}_I$ and an element theory $\mathcal{T}_E$. The signature of $\mathcal{T}_A$ consists of the two functions $\cdot[\cdot]$ and $\cdot\langle\cdot \lhd \cdot\rangle$. Every model of the theory of arrays satisfies the select-over-store-axioms proposed by McCarthy [7]:

$$\forall a\, i\, v.\ a\langle i \lhd v\rangle[i] = v \tag{idx}$$

$$\forall a\, i\, j\, v.\ i \neq j \implies a\langle i \lhd v\rangle[j] = a[j] \tag{read-over-write}$$

Additionally we consider the extensional variant of the theory of arrays. Then, every model has to satisfy the extensionality axiom:

$$\forall a\, b.\ a = b \lor \exists i.\ a[i] \neq b[i] \tag{ext}$$

We use a, b to denote array-valued variables, i, j, k to denote index variables, and v, w to denote element variables. Additionally we use subscripts to distinguish different variables. We use P to denote a path in a graph. A path in a graph is interpreted as a sequence of edges.

In the remainder of this paper, we consider quantifier-free $\mathcal{T}_A$-formulae. Furthermore we fix the index $\mathcal{T}_I$ to a stably infinite theory and the element theory $\mathcal{T}_E$ to a theory that contains at least two different values[1].

3 Towards a Nelson–Oppen-based Array Solver

Multiple theories are usually combined with a variant of the Nelson–Oppen combination procedure [9]. The procedure requires the participating theories to be stably infinite and to only share the equality symbol =.

The procedure first transforms the input such that every literal is *pure* with respect to the theories. Let $f(t)$ be a term in the input. If f is interpreted by theory $\mathcal{T}_1$ and t is interpreted by theory $\mathcal{T}_2$, then $f(t)$ is not pure. The first step of the Nelson–Oppen procedure then generates a fresh variable v, rewrites $f(t)$ into $f(v)$, and adds the definition $v = t$ as a new conjunct to the formula. The fresh variable is shared between theories $\mathcal{T}_1$ and $\mathcal{T}_2$. This step is repeated until all terms are pure. By abuse of notation, we name the shared variable after its defining term t, e. g., we use $a[i]$ to denote the shared variable that is defined as $a[i]$.

Let V be the set of fresh variables introduced in the first step of the combination procedure. The second step of the procedure tries to find an *arrangement* of V, i. e., an equivalence relation between variables in V such that $\mathcal{T}_1$ and $\mathcal{T}_2$ produce partial models that agree with this equivalence relation. Finding such an arrangement is typically done by propagating equalities or providing case split lemmas. In the following, we call this arrangement strong equivalence to distinguish it from weak equivalence defined in the next section. We write $a \sim b$ to denote that a and b are strongly equivalent, i.e., that in the current arrangement the shared variables a and b are equal.

For the theory of arrays, we consider every term of the form $\cdot\langle\cdot \lhd \cdot\rangle$ or $\cdot[\cdot]$ as being interpreted by the array theory. We consider all array terms, store, and select terms to be shared and thus they have to occur in the arrangement. Furthermore, every index term that appears in a store or select is considered shared between the array theory and the index theory. Then the goal is to find a suitable arrangement to these shared terms such that all theories agree on this arrangement.

For an array solver to be used in Nelson–Oppen combination we have to propagate equalities between shared array terms and shared select terms. Furthermore, the other theories have to propagate equalities between terms used as index in a select or store. In the remainder of this paper we will first show how to propagate equalities between select terms and afterwards deal with extensionality to propagate equalities between array-valued terms.

4 Weak Equivalences over Arrays

The theory of arrays has two constructors for arrays: array variables, and store terms $\cdot\langle \cdot \lhd \cdot\rangle$. Assuming quantifier-free input, we can only constrain the values of a finite number of indices.

[1] Note that $\mathcal{T}_A$ is stably infinite under these conditions.

These constraints can either be explicity like $a[i] = v$, or implicit like $a\langle i \lhd v\rangle$ where axiom (idx) produces the corresponding $a\langle i \lhd v\rangle[i] = v$. Hence, for quantifier-free input, arrays that are connected via a sequence of $\cdot\langle \cdot \lhd \cdot\rangle$ can only differ in finitely many positions. We call such arrays *weakly equivalent*. In this section we formally define weak equality and show how to exploit this to produce a decision procedure for the (extensional) theory of arrays.

Let $\mathcal{S}$ be the set of all terms of the form $\cdot\langle \cdot \lhd \cdot\rangle$ in the input formula and $\mathcal{A}$ be the set of all array-valued terms that are not in $\mathcal{S}$. Since $a\langle i \lhd v\rangle$ modifies a only at index i, these two arrays are guaranteed to be equal on all indices except on index i. We generalise this observation to chains of the form $\ldots\langle j \lhd w\rangle\langle i \lhd v\rangle$ to extract a set of indices for which two arrays might store different values.

Definition 1 (weak equivalence)**.** A *weak equivalence graph* G^W contains vertices $\mathcal{S} \cup \mathcal{A}$ and undirected edges defined as follows:

1. $a \leftrightarrow b$ if $a \sim b$, and
2. $a \overset{i}{\leftrightarrow} b$ if a has form $b\langle i \lhd \cdot\rangle$.

We write $a \overset{(P)}{\Leftrightarrow} b$ if there exists a path P between nodes a and b in G^W. In this case, we call a and b *weakly equal*. The weak equivalence class containing all elements that are weakly equal to a is defined as $\text{WeakEQ}(a) := \{b \mid \exists P.\ a \overset{(P)}{\Leftrightarrow} b\}$.

For a path P we define $\text{Stores}\,(P)$ as the set of all indices corresponding to edges of the form $\overset{\cdot}{\leftrightarrow}$, i. e., $\text{Stores}\,(P) := \{i \mid \exists a\, b.\ a \overset{i}{\leftrightarrow} b \in P\}$.

Example 1. Consider the formula $a = b\langle j \lhd v\rangle \wedge b = c\langle i \lhd w\rangle \wedge d = e \wedge c[i] = w$. The weak equivalence graph for this example is shown in Figure 1. Note that the last conjunct is not important for the construction of the weak equivalence graph.

$$\begin{tikzcd}
a \arrow[r, leftrightarrow] & b\langle j \lhd v\rangle \arrow[r, leftrightarrow, "j"] & b \arrow[r, leftrightarrow] & c\langle i \lhd w\rangle \arrow[r, leftrightarrow, "i"] & c \\
& & d \arrow[r, leftrightarrow] & e &
\end{tikzcd}$$

Figure 1: Weak Equivalence Graph for Example 1

We get two different weak equivalence classes. The first one contains the nodes a, $b\langle j \lhd v\rangle$, b, $c\langle i \lhd w\rangle$, and c. The second contains d and e. Note that d and e are actually strongly equivalent. Thus, they store the same value at every position. Let P denote the path from a to c in the weak equivalence graph. Then, $\text{Stores}\,(P) = \{i, j\}$. Thus, arrays a and c can only differ in at most the values stored at the indices i and j.

If we want to know if $a[i]$ and $b[i]$ should be equal, we check if $a \overset{(P)}{\Leftrightarrow} b$ for a path P such that $i \notin \text{Stores}\,(P)$. If this is the case, P witnesses the equivalence between the select terms.

Definition 2 (weak equivalence modulo i)**.** Two arrays a and b are *weakly equivalent modulo* i if and only if they are weakly equivalent and connected by a path that does not contain an edge of the form $\overset{j}{\leftrightarrow}$ where $j \sim i$. We denote weak equivalence modulo i by $a \approx_i b$ and define it as $a \approx_i b := \exists P.\ a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in \text{Stores}\,(P)\,.\ j \not\sim i$.

Using this definition we can propagate equalities between shared selects if the arrays are weakly equivalent modulo the index of the select.

Lemma 1 (read-over-weakeq)**.** *Let $\sim$ be an arrangement satisfying the array axioms. Let $a[i]$ and $b[j]$ be two selects such that $i \sim j$ and $a \approx_i b$. Then, $a[i] \sim b[j]$ holds.*

Proof. We induct over the length of the path P witnessing $a \approx_i b$.

Base case. In this case, a and b are the same term. Hence $a[i] \sim b[j]$ holds by congruence.

Step case. Let the step from c to b be the last step of path P. By induction hypothesis we know that $a[i] \sim c[j]$ holds.

If the edge between c and b is due to a strong equivalence (i. e., $c \sim b$), then $c[j] \sim b[j]$ follows from congruence.

If the edge between c and b is of the form $c \overset{k}{\leftrightarrow} b$, then either c is $b\langle k \lhd \cdot\rangle$ or b is $c\langle k \lhd \cdot\rangle$. In both cases, we get the lemma $j = k \vee c[j] = b[j]$ from axiom (read-over-write). Since $j \sim i$ and $i \not\sim k$, we get $j \not\sim k$ and thus $c[j] \sim b[j]$. We conclude $a[i] \sim b[j]$ by transitivity. □

This lemma allows us to propagate equalities between shared selects. Note that it depends upon disequalities between index terms needed to ensure $a \approx_i b$.

If two arrays are weak equivalent modulo i they store the same value at the index i. The reverse is not necessarily true. Therefore, we define a weaker relation weak congruence modulo i.

Definition 3 (weak congruence modulo i)**.** Arrays a and b are *weak congruent modulo* i if and only if they are guaranteed to store the same value at index i. We denote weak congruence modulo i by $\sim_i$ and define $a \sim_i b := a \approx_i b \vee \exists a'\, b'\, j\, k.\ a \approx_i a' \wedge i \sim j \wedge a'[j] \sim b'[k] \wedge k \sim i \wedge b' \approx_i b$.

We use weak congruences to decide extensionality. Intuitively, if for all indices i the weak congruence modulo i $a \sim_i b$ holds, then $a = b$ should be propagated. But this naïve approach requires checking every index occurring in the formula. To minimise the number of indices we need to consider, we exploit the weak equivalence graph.

Lemma 2 (weakeq-ext)**.** *Let $\sim$ be an arrangement satisfying the array axioms. Let a and b be two arrays such that $a \overset{(P)}{\Leftrightarrow} b$ holds. If for all indices $i \in \mathrm{Stores}(P)$ we have $a \sim_i b$, then $a \sim b$ holds.*

Proof. Follows from Lemma 1, Definition 3 and (ext). □

5 A Decision Procedure Based on Weak Equivalences

Our decision procedure is based on weak equivalences and the Nelson–Oppen combination scheme. It propagates equalities between terms shared by multiple theories. We limit the propagation to shared array terms and array select terms.

The $\mathcal{T}_A$-formulae are preprocessed as follows. For every $a\langle i \lhd v\rangle$ contained in the input, we (1) instantiate the axiom (idx) and (2) add $a[i]$ to the set of terms contained in the input[2]. Thus, the preprocessing step adds at most two select operations for every store.

We propagate new equalities from weak equivalence relations and weak congruence relations based on lemmas 1 and 2. These relations depend on the arrangement $\sim$, which represents logical equality ($=$). We now define a function $\mathrm{Cond}(\cdot)$ that computes a condition (a conjunction of equalities and inequalities) under which a weak equivalence or weak congruence holds. To

[2] This can be achieved by adding the equality $a[i] = a[i]$.

denote the condition for a path that does not contain an edge labelled with index i we use $\mathrm{Cond}_i(\cdot)$. For an edge in the weak equivalence graph that represents an equality, the condition reflects this equality. For an edge that comes from a $\cdot\langle j \lhd \cdot\rangle$, no condition is needed. However, $\mathrm{Cond}_i(\cdot)$ should ensure that i does not occur on the path, so $i \neq j$ needs to hold.

$$\mathrm{Cond}(a \leftrightarrow b) := a = b \qquad \mathrm{Cond}_i(a \leftrightarrow b) := a = b$$

$$\mathrm{Cond}(a \overset{j}{\leftrightarrow} b) := \mathrm{true} \qquad \mathrm{Cond}_i(a \overset{j}{\leftrightarrow} b) := i \neq j$$

We can extend these definitions to paths by conjoining the conditions for all edges on that path. Then, we can compute $\mathrm{Cond}(a \approx_i b)$ using the path that witnesses $a \approx_i b$.

$$\mathrm{Cond}(a \approx_i b) := \mathrm{Cond}_i(P) \text{ where } a \overset{(P)}{\Leftrightarrow} b \wedge \forall j \in \mathrm{Stores}\,(P)\,.\; i \not\sim j$$

Finally, to define $\mathrm{Cond}(a \sim_i b)$, we use the definition of $\sim_i$.

$$\mathrm{Cond}(a \sim_i b) := \begin{cases} \mathrm{Cond}(a \approx_i b) & \text{if } a \approx_i b \\ \quad \mathrm{Cond}(a \approx_i a') \wedge i = j \wedge a'[j] = b'[k] & \text{if } \begin{array}{c} a \approx_i a' \wedge i \sim j \wedge a'[j] \sim b'[k] \\ \wedge\, k \sim i \wedge b' \approx_i b \end{array} \\ \qquad \wedge\, k = i \wedge \mathrm{Cond}(b' \approx_i b) & \end{cases}$$

Example 2. Consider again the formula $a = b\langle j \lhd v\rangle \wedge b = c\langle i \lhd w\rangle \wedge d = e \wedge c[i] = w$ from Example 1 whose weak equivalence graph is shown in Figure 1. Assume $i \not\sim j$. Then we have $a \approx_i c\langle i \lhd w\rangle$ since no edge contains a label that is equivalent to i. We get $\mathrm{Cond}(a \approx_i c\langle i \lhd w\rangle) \equiv a = b\langle j \lhd v\rangle \wedge i \neq j \wedge b = c\langle i \lhd w\rangle$.

From Axiom (idx) we get $c\langle i \lhd w\rangle[i] = w$. With $c[i] = w$ we conclude $a \sim_i c$ since $a \approx_i c\langle i \lhd w\rangle$ and $c\langle i \lhd w\rangle[i] = c[i]$. We have $\mathrm{Cond}(a \sim_i c) \equiv \mathrm{Cond}(a \approx_i c\langle i \lhd w\rangle) \wedge c\langle i \lhd w\rangle[i] = c[i]$.

To decide the theory of arrays we define two rules to generate instances of array lemmas. We present the rules as inference rules. The rule is applicable if the current arrangement $\sim$ on the shared variables V satisfies the conditions above the line. The rule then generates a new (valid) lemma that can propagate an equality under the current arrangement.

The first rule is based on Lemma 1. Two select terms are equivalent if the indices of the selects are congruent and the arrays are weakly equivalent modulo that index. We only create this lemma if the select terms existed in the formula. Note that we create for select terms in the formula a shared variable with the same name in V.

$$\frac{a \approx_i b \qquad i \sim j \qquad a[i], b[j] \in V}{i \neq j \vee \neg\,\mathrm{Cond}(a \approx_i b) \vee a[i] = b[j]} \qquad \text{(read-over-weakeq)}$$

The next rule is based on Lemma 2 and used to propagate an equality between two extensionally equal array terms. Two arrays a and b have to be equal if there is a path P such that $a \overset{(P)}{\Leftrightarrow} b$ and for all $i \in \mathrm{Stores}\,(P)$, $a \sim_i b$ holds.

$$\frac{a \overset{(P)}{\Leftrightarrow} b \qquad \forall i \in \mathrm{Stores}\,(P)\,.\; a \sim_i b \qquad a, b \in V}{\neg\,\mathrm{Cond}(P) \vee \bigvee_{i \in \mathrm{Stores}(P)} \neg\,\mathrm{Cond}(a \sim_i b) \vee a = b} \qquad \text{(weakeq-ext)}$$

The resulting decision procedure is sound and complete for the existential theory of arrays assuming sound and complete decision procedures for the index and element theories.

Lemma 3 (soundness)**.** *Rules (read-over-weakeq) and (weakeq-ext) are sound.*

Proof. Soundness of the rules follows directly from the lemma with the corresponding name. □

Lemma 4 (completeness)**.** *The rules (read-over-weakeq) and (weakeq-ext) are complete.*

Proof. Assume all rules are saturated. Let $\mathcal{M}$ be the model generated by the theories different from the array theory. In this model, arrays are considered uninterpreted and only subject to congruence. Note that $v_1 \sim v_2$ if and only if $\mathcal{M}(v_1) = \mathcal{M}(v_2)$ is guaranteed in this model. We create a new model $\mathcal{M}_A$ that extends $\mathcal{M}$ by the interpretation of the array terms in the formula.

First, for every array type $\sigma \Rightarrow \tau$, we define its domain as the set of all functions from σ to τ. The interpretation of $\cdot[\cdot]$ is function application and the interpretation of $\cdot\langle\cdot \lhd \cdot\rangle$ is function update. This definition trivially satisfies the array axioms (idx) and (read-over-write). For every other function symbol f and every constant v that is not of array type we define $\mathcal{M}_A(f) = \mathcal{M}(f)$ and $\mathcal{M}_A(v) = \mathcal{M}(v)$. Next we define the interpretation for all constants of type array in $\mathcal{M}_A$, such that $\mathcal{M}_A$ satisfies the input formula.

Let $\prec$ be a partial order on types such that for every type $\sigma \Rightarrow \tau$ we have $\sigma \prec \sigma \Rightarrow \tau$ and $\tau \prec \sigma \Rightarrow \tau$. We define the interpretation of constants according to $\prec$. Thus, when defining the interpretation for a constant of type $\sigma \Rightarrow \tau$ we assume all constants of type σ resp. τ are already defined.

For each sort τ we assume two different values Fst_τ and Snd_τ. Furthermore we assume every sort σ that is used as index sort for an array sort contains an infinite supply of fresh domain elements denoted by $\ulcorner\cdot\urcorner$. Then, for an array constant a of sort $\sigma \Rightarrow \tau$, we define

$$\mathcal{M}_A(a)(\jmath) := \begin{cases} \mathcal{M}(b[i]) & \text{if } b[i] \text{ occurs in input, } \mathcal{M}(i) = \jmath \text{ and } a \approx_i b \\ \mathit{Snd}_\tau & \text{if } \jmath = \ulcorner \text{WeakEQ}(a) \urcorner \\ \mathit{Fst}_\tau & \text{otherwise} \end{cases}$$

The first case is well defined. Given two select terms $b_1[i_1]$ and $b_2[i_2]$ with $a \approx_{i_1} b_1$, $a \approx_{i_2} b_2$ and $\mathcal{M}(i_1) = \jmath = \mathcal{M}(i_2)$ we need to show $\mathcal{M}(b_1[i_1]) = \mathcal{M}(b_2[i_2])$. Since $\mathcal{M}(i_1) = \mathcal{M}(i_2)$, we have $i_1 \sim i_2$ and $b_1 \approx_{i_1} b_2$ (since $\approx_{i_1}$ and $\approx_{i_2}$ is the same relation). The rule (read-over-weakeq) generated the lemma $i_1 \neq i_2 \vee \neg\,\text{Cond}(b_1 \approx_{i_1} b_2) \vee b_1[i_1] = b_2[i_2]$. $\mathcal{M}$ guarantees that $i_1 = i_2$ and $\text{Cond}(b_1 \approx_{i_1} b_2)$ hold. Thus, $\mathcal{M}(b_1[i_1]) = \mathcal{M}(b_2[i_2])$ has to hold in order to satisfy this lemma.

We have to show that the input formula is satisfied by $\mathcal{M}_A$ if it is satisfied by $\mathcal{M}$. We assume that the array operations in the input formula were flattened by introducing fresh variables, i. e., that $\cdot[\cdot]$ and $\cdot\langle\cdot \lhd \cdot\rangle$ occur only in definitions $v = a[i]$ and $b = a\langle i \lhd v\rangle$. By definition of $\mathcal{M}_A$ it already satisfies the parts of the input formulae that do not involve arrays. It remains to show that

1. for all definition $v = a[i]$ $\mathcal{M}_A(v) = \mathcal{M}_A(a[i])$ holds.

2. for all definition $b = a\langle i \lhd v\rangle$ $\mathcal{M}_A(b) = \mathcal{M}_A(a\langle i \lhd v\rangle)$ holds.

3. $\mathcal{M}_A(a) = \mathcal{M}_A(b)$ if and only if $\mathcal{M}(a) = \mathcal{M}(b)$ for array constants a and b, and

1. For a definition $v = a[i]$ the model $\mathcal{M}$ already guarantees $\mathcal{M}(v) = \mathcal{M}(a[i])$. The definition of $\mathcal{M}_A$ gives us $\mathcal{M}_A(v) = \mathcal{M}(v) = \mathcal{M}(a[i]) = \mathcal{M}_A(a)(\mathcal{M}(i)) = \mathcal{M}_A(a[i])$ as required.

2. For $b = a\langle i \lhd v\rangle$ we need to show $\mathcal{M}_A(b)(\jmath) = \mathcal{M}_A(a)(\jmath)$ for $\jmath \neq \mathcal{M}(i)$ and $\mathcal{M}_A(b)(\mathcal{M}(i)) = \mathcal{M}(v)$. Our preprocessing step adds the equality $a\langle i \lhd v\rangle[i] = v$ to the input. Thus, $\mathcal{M}_A(b)(\mathcal{M}(i)) = \mathcal{M}(a\langle i \lhd v\rangle[i]) = \mathcal{M}(v)$. For $\jmath \neq \mathcal{M}(i)$, we can derive $\mathcal{M}_A(b)(\jmath) = \mathcal{M}_A(a)(\jmath)$ from $\text{WeakEQ}(b) = \text{WeakEQ}(a)$ and the fact that $a \approx_j b$ for every $j \not\sim i$.

3. We show that our extended model satisfies equalities on arrays. If $a \sim b$ holds for two arrays a and b, then by construction $\mathcal{M}_A(a) = \mathcal{M}_A(b)$. If $a \not\sim b$ holds for two arrays a and b, then we distinguish two cases. If a and b are not connected in the weak equivalence graph, they differ at the indices $\ulcorner\text{WeakEQ}(a)\urcorner$ and $\ulcorner\text{WeakEQ}(b)\urcorner$. Hence, $\mathcal{M}_A(a) \neq \mathcal{M}_A(b)$ holds. Otherwise, since rule (weakeq-ext) is saturated and $a \overset{(P)}{\Leftrightarrow} b$ for some path P but $a \neq b$, there exists an $i \in \text{Stores}\,(P)$ such that $a \not\sim_i b$ holds. Let a' be the first array on the path P that involves an edge $\overset{i}{\leftrightarrow}$ and b' the last such array. In the preprocessing step we added $a'[i]$ and $b'[i]$. By the choice of a' and b' we have $a \approx_i a'$ and $b' \approx_i b$. Since $a \not\sim_i b$ holds, we have $a'[i] \not\sim b'[i]$. By definition $\mathcal{M}_A(a)(\mathcal{M}(i)) = \mathcal{M}(a'[i])$ and $\mathcal{M}_A(b)(\mathcal{M}(i)) = \mathcal{M}(b'[i])$, so $\mathcal{M}_A(a)$ and $\mathcal{M}_A(b)$ are different arrays as desired.

□

6 Restricting Instantiations

The preprocessor is the only component of our decision procedure that produces new select terms and thus might trigger new lemmas. These lemmas only generate new (dis-)equality literals between existing terms. Thus, reducing the number of select terms might reduce the number of lemmas generated by our decision procedure and speed up the procedure.

If the element theory is stably infinite we can omit the preprocessor step that adds for every $a\langle i \lhd v\rangle$ the select $a[i]$. Instead, we simply assume $a[i]$ to be different than any other $b[i]$ unless $a \sim b$. This method preserves soundness and completeness.

Lemma 5. *(soundness of modified procedure) The modified procedure is sound.*

Proof. Follows directly from Lemma 3 since it does not rely on the addition of $a[i]$ for every $a\langle i \lhd \cdot\rangle$. □

For the completeness lemma we take into account the fact that the element theory is stably infinite. Thus, if $a[i]$ is not present we use a fresh element in the value domain.

Lemma 6 (completeness of modified procedure)**.** *The modified procedure is complete.*

Proof. We redefine the generation of the model $\mathcal{M}_A$ in the proof of Lemma 4. Let $\text{WeakEQ}_i(a) := \{b \mid b \approx_i a\}$ denote the set of all array terms that are weakly equivalent modulo i to a. We construct $\mathcal{M}_A$ in the following way.

$$\mathcal{M}_A(a)(\jmath) := \begin{cases} \mathcal{M}(b[i]) & \text{if } b[i] \text{ occurs in the input, } \mathcal{M}(i) = \jmath \text{ and } a \approx_i b \\ \ulcorner\text{WeakEQ}_i(a)\urcorner & \text{if } \mathcal{M}(i) = \jmath, i \in \text{Stores}\,(\text{WeakEQ}(a)) \text{ and there are no } b \text{ and } j \\ & \quad \text{such that } b[j] \text{ occurs in the input, } \mathcal{M}(j) = \jmath \text{ and } a \approx_j b \\ \mathit{Snd}_\tau & \text{if } \jmath = \ulcorner\text{WeakEQ}(a)\urcorner \\ \mathit{Fst}_\tau & \text{otherwise} \end{cases}$$

Note that we only change the model for array a and index i if $a\langle i \lhd \cdot\rangle$ exists in the input, but $a[i]$ does not. Hence, for definition of the form $v = a[i]$ and $b = a\langle i \lhd v\rangle$ the old proof can be reused. Thus, we only need to show extensionality.

If $\mathcal{M}(a) = \mathcal{M}(b)$ for two array terms a and b, then $a \sim b$. Thus, for all indices i, $\mathcal{M}_A(a)(i) = \mathcal{M}_A(b)(i)$ holds by construction. Otherwise we distinguish two cases. If a and b are not connected in the weak equivalence graph, $\mathcal{M}_A(a)$, $\mathcal{M}_A(b)$ differ, e. g., on index $\ulcorner\text{WeakEQ}(a)\urcorner$. If the arrays are connected ($a \overset{(P)}{\Leftrightarrow} b$), there is an $i \in \text{Stores}\,(P)$ such that $a \not\approx_i b$. If there is no $a'[j]$ with $a \approx_i a'$ and $j \sim i$, then $\mathcal{M}_A(a)(\mathcal{M}(i)) = \ulcorner\text{WeakEQ}_i(a)\urcorner$ differs from $\mathcal{M}_A(b)(\mathcal{M}(i))$. Similarly for b, if there is no $b'[k]$ with $b \approx_i b'$ and $k \sim i$. Otherwise $\mathcal{M}_A(a)(\mathcal{M}(i)) = \mathcal{M}(a'[j])$ and $\mathcal{M}_A(b)(\mathcal{M}(i)) = \mathcal{M}(b'[k])$ and these values differ since $a \not\approx_i b$. □

This optimisation enables us to limit the number of additional terms in the input. Since we only need to generate (read-over-weakeq) lemmas if the select terms in the conclusion are present after preprocessing, this optimisation also reduces the number of lemmas. Furthermore, it is widely applicable. In fact, the non-bitvector logics defined in the SMTLIB [2] only allow array sorts where the element theory is stably infinite. Thus, only the terms corresponding to instantiations of Axiom (idx) are required. In an actual implementation even these terms could be omitted (see [3]).

7 Implementation and Evaluation

We implemented the decision procedure described in this paper in our SMT solver SMTInterpol [5]. Besides the aforementioned preprocessing step that applies (idx) to every $\cdot\langle\cdot \lhd \cdot\rangle$ in the input, we also simplify $\mathcal{T}_A$-formulae by applying (read-over-write) if the index of the store and the index of the select are syntactically equal. Furthermore, we contract terms of the form $a\langle i \lhd v_2\rangle\langle i \lhd v_1\rangle$ to $a\langle i \lhd v_1\rangle$. We only add $a[i]$ to the set of terms contained in the formula if we have $a\langle i \lhd v\rangle$ in the input and the domain of v is finite.

We represent the weak equivalence relation and the weak equivalence modulo i relations in a forest structure, similarly to the representation of equivalence graph in congruence solvers [10]. Every node has an outgoing edge, and these edges build a spanning tree for every equivalence class. The edges point from a child node to the parent node. The root node of every tree has no outgoing edge and is the representative of its equivalence class.

We have to distinguish between strong equivalence, weak equivalence, and weak equivalence modulo i. The strong equivalence classes are already handled by the equality solver. In our implementation of the array solver we treat them as indivisible and create a single node for every strong equivalence class. To represent the weak equivalence relations the nodes have up to two outgoing edges, a primary p and a secondary s, see Figure 2. The edges come from a store operation and correspond to the edges $\overset{i}{\leftrightarrow}$ in the weak equivalence graph. The index of the primary edge is stored in the *pi* field. The primary edge points towards the representative of the weak equivalence class. Every primary edge p connects the node representing (the strong equivalence class of) a store $a\langle j \lhd v\rangle$ with the node representing a and the corresponding index in the *pi* field is j. Note, however, that the direction of the edge can be arbitrary, as we invert the edges during the execution of the algorithm. If the primary edge is missing the node is the representative of its weak equivalence class and of all its weak equivalence modulo i classes.

While the primary edge is enough to represent the weak equivalence relation we need another edge to represent weak equivalence modulo i. The representative of weak equivalence modulo i is also found by following the primary edges. However, if the store of the primary edge is on the index i, the secondary edge is followed instead. If the secondary edge is missing the node is the representative of its weak equivalence modulo i class.

The equivalence classes are represented as follows. Two arrays a and b are weakly equivalent iff $\text{GET-REP}(a) = \text{GET-REP}(b)$ and $a \approx_i b$ iff $\text{GET-REP}_i(a, i) = \text{GET-REP}_i(b, i)$.

```
struct NODE
    $p$ : NODE
    $pi$ : INDEX
    $s$ : NODE

GET-REP($n$ : NODE)
  if $n.p$ = NIL then $n$
  else      GET-REP($n.p$)

MAKE-REP($n$ : NODE)
  if $n.p \neq$ NIL then
      MAKE-REP($n.p$)
      $n.p.p := n$           } invert primary
      $n.p.pi := n.pi$       } edge
      $n.p :=$ NIL           }
      MAKE-REP$_i$($n$)
```

```
GET-REP$_i$($n$ : NODE, $i$ : INDEX)
  if $n.p$ = NIL then $n$
  elseif $n.pi \neq i$ then GET-REP$_i$($n.p, i$)
  elseif $n.s$ = NIL then n
  else      GET-REP$_i$($n.s, i$)

MAKE-REP$_i$($n$ : NODE)
  if $n.s \neq$ NIL then
      if $n.s.pi \neq n.pi$ then
          $n.s := n.s.p$       move towards
                               representative
          MAKE-REP$_i$($n$)
      else
          MAKE-REP$_i$($n.s$)
          $n.s.s := n.s$       } invert
          $n.s :=$ NIL         } secondary edge
```

Figure 2: Data structure and functions to represent weak equivalence relations. A NODE structure is created for every strong equivalence class on arrays. It contains two outgoing edges p, s pointing towards the representative of the weak equivalence classes. The functions GET-REP and GET-REP$_i$ are used to find the representative of the weak equivalence (resp. weak equivalence modulo i) class. The functions MAKE-REP and MAKE-REP$_i$ invert the edges to make a node the representative of its weak equivalence classes.

```
ADD-SECONDARY($S$ : INDEX SET, $a, b$ : NODE)
  if $a = b$ then
      return
  if $a.pi \notin S \wedge$ GET-REP$_i$($a, a.pi$) $\neq b$ then
      MAKE-REP$_i$($a$)
      $a.s := b$
  ADD-SECONDARY($S \cup \{a.pi\}, a.p, b$)
```

```
ADD-STORE($a, b$ : NODE, $i$ : INDEX)
  MAKE-REP(b)
  if GET-REP($a$) = $b$ then
      ADD-SECONDARY($\{i\}, a, b$)
  else
      $b.p := a$
      $b.pi := i$
```

Figure 3: The algorithm ADD-STORE adds a new store edge to the data structure updating the weak equivalence classes. In the else case a new primary edge is added to merge two disjoint weak equivalence classes. Otherwise, ADD-SECONDARY inserts new secondary edges to merge the necessary weak equivalence modulo i classes.

The algorithm proceeds by inserting the store edges one by one, similarly to the algorithm presented in [10]. The algorithm that inserts a store edge is given in Figure 3. The algorithm first inverts the outgoing edges of one node to make it the representative of its weak equivalence class. If the other side of the store edge lies in a different weak equivalence classes, the store can be inserted as a new primary edge.

If the nodes are already weakly equivalent the procedure ADD-SECONDARY is called. This procedure follows the path from the other array a to the array b that was made the representative. For every node on this path it checks if a secondary edge needs to be added. If the primary edge of the node is labelled with a store on i, the algorithm first checks if the node is weakly equivalent modulo i with b due to the new store edge. This is the case if no store on i occurred on the path so far and the new store is also on an index different from i. We use the set S to collect these forbidden indices. Then if b is not already the representative of the weak equivalence modulo i class, the outgoing secondary edges are reversed and a new secondary edge is added.

The complexity of the procedure ADD-STORE is worst case quadratic in the size of the weak equivalence class. This stems from MAKE-REP$_i$ being linear in the size and being called a linear number of times. The overall complexity is cubic in the number of stores in the input formula. The space requirement, however, is only linear. In our current implementation in SMTInterpol this procedure was not a bottleneck so far. In SMTInterpol we also keep the stores that created the primary and secondary edge in the data-structure. This allows for computing the paths needed for lemma generation in linear time.

Example 3. Figure 4 shows an example of the data structure where the primary edges are labelled by the index of the corresponding store. This data structure represents only one weak equivalence class with the representative node 0. The resulting data structure after adding a store with index k between nodes 0 and 4 is shown on the right. Since nodes 0 and 4 were already in the same weak equivalence class, secondary edges were added.

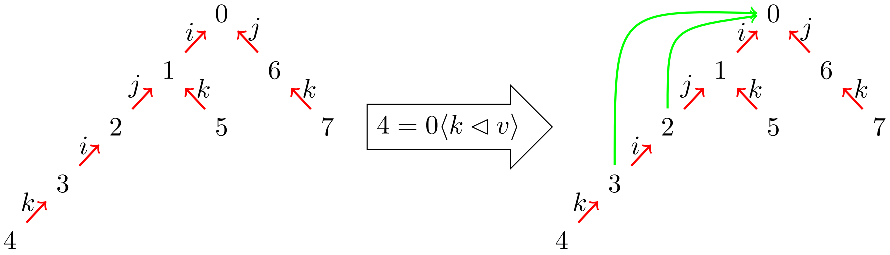
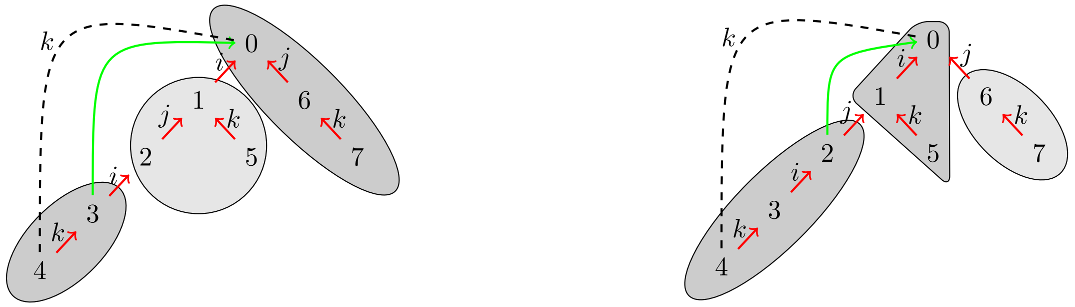


Figure 4: Weak equivalence classes represented by a graph using primary and secondary edges. The short direct edges are primary edges, the long bended edges are secondary edges. Each primary edge represents a store edge between the connected nodes and is labelled by the index of the store. The secondary edges in the right graph were created by a store edge on index k between node 0 and 4 as described in Example 3.

These secondary edges are needed to connect the weak equivalence modulo i and modulo j classes. Figure 5(a) shows how the first secondary edge connects the two weak equivalence modulo i classes rooted at nodes 0 resp. 3. This is necessary since there is now a new path using the edge from 4 to 0. Note that no secondary edge is added to node 1, since nodes 1, 2, and 5 are still not weakly equivalent modulo i to the other nodes. Figure 5(b) shows the connection between the two weak equivalence modulo j classes rooted at nodes 0 resp. 2. The weak equivalence modulo j class rooted at node 6 is not affected by a new edge between nodes 0 and 4 since these nodes are on a different path.

We implemented this decision procedure in our SMT solver SMTInterpol [5] and tested it on the benchmarks from the QF_AX and QF_AUFLIA devisions of the SMTEVAL 2013 benchmarks. We solved all benchmarks in 1:32 resp. 10:45 minutes without running into a timeout of 10 minutes. According to the data from the SMTEVAL, no other solver was able to solve all benchmarks in these divisions. We defer an up-to-date comparison to the SMTCOMP 2014.



(a) Merging weak equivalence modulo i classes.

(b) Merging weak equivalence modulo j classes.

Figure 5: Secondary edges merge weak equivalence modulo i classes.

8 Conclusion and Future Work

We presented a new decision procedure for the extensional theory of arrays. This procedure exploits weak equalities to limit the number of axiom instantiations. The instantiations produced by the decision procedure presented in this paper can be restricted to terms already present in the input formula. Furthermore we discussed an implementation based on a graph structure similar to congruence closure graphs. This decision procedure is implemented in our SMT solver SMTInterpol [5]. We plan to implement a variant of the quantifier-free interpolation for arrays [4] based on the lemmas generated by this decision procedure. Since these lemmas only generate mixed equalities, proof tree preserving interpolation [6] can be used.

References

[1] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)

[2] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: 2.0. In: SMT (2010)

[3] Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. JSAT 6(1-3), 165–201 (2009)

[4] Bruttomesso, R., Ghilardi, S., Ranise, S.: Quantifier-free interpolation of a theory of arrays. Logical Methods in Computer Science 8(2) (2012)

[5] Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: SPIN. pp. 248–254 (2012)

[6] Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. In: TACAS. pp. 124–138 (2013)

[7] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. pp. 21–28 (1962)

[8] de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD. pp. 45–52 (2009)

[9] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)

[10] Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: RTA. pp. 453–468. Springer (2005)