7

Arrays

7.1 Introduction

The array is a basic datatype that is supported by most programming languages, and is consequently prevalent in software. It is also used for modeling the memory components of hardware. It is clear, then, that analysis of software or hardware requires the ability to decide formulas that contain arrays. This chapter introduces an array theory and two decision procedures for specific fragments thereof.

Let us begin with an example that illustrates the use of array theory for verifying an invariant of a loop.

Example 7.1. Consider the pseudocode fragment in Fig. 7.1. The main step of the correctness argument is to show that the assertion in line 7 follows from the assertion in line 5 when executing the assignment in line 6. A common way to do so is to generate **verification conditions**, e.g., using Hoare's axiom system. We obtain the following verification condition for the claim:

$$\begin{array}{rl} & (\forall x \in \mathbb{N}_0.\ x < i \implies \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}' = \mathtt{a}\{i \leftarrow 0\} \\ \implies & (\forall x \in \mathbb{N}_0.\ x \leq i \implies \mathtt{a}'[x] = 0)\ . \end{array} \tag{7.1}$$

The formula above contains two symbols that are specific to arrays: the *array index* operator $\mathtt{a}[x]$ and the *array update* operator $\mathtt{a}\{i \leftarrow 0\}$. We will explain the meaning of these operators later. The validity of (7.1) implies that the loop invariant is maintained. Our goal is to prove such formulas automatically, and indeed later in this chapter we will show how this can be done. ⌟

The array theory permits expressions over arrays, which are formalized as maps from an *index type* to an *element type*. We denote the index type by T_I, and the element type by T_E. The type of the arrays themselves is denoted by T_A, which is a shorthand for $T_I \longrightarrow T_E$, i.e., the set of functions that map an

T_I T_E T_A

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_7

```
1  a: array 0..99 of integer;
2  i: integer;
3
4  for i:=0 to 99 do
5          assert(∀x ∈ ℕ0. x < i ⟹ a[x] = 0);
6          a[i]:=0;
7          assert(∀x ∈ ℕ0. x ≤ i ⟹ a[x] = 0);
8  done;
9  assert(∀x ∈ ℕ0. x ≤ 99 ⟹ a[x] = 0);
```

Fig. 7.1. Pseudocode fragment that initializes an array of size 100 with zeros, annotated with assertions

element of T_I to an element of T_E. Note that neither the set of indices nor the set of elements are required to be finite.

Let $a \in T_A$ denote an array. There are two basic operations on arrays:

$a[i]$

1. *Reading* an element with index $i \in T_I$ from a. The value of the element that has index i is denoted by $a[i]$. This operator is called the *array index* operator.

$a\{i \leftarrow e\}$

2. *Writing* an element with index $i \in T_I$. Let $e \in T_E$ denote the value to be written. The array a where element i has been replaced by e is denoted by $a\{i \leftarrow e\}$. This operator is called the *array update* or *array store* operator.

We call the theories used to reason about the indices and the elements the *index theory* and the *element theory*, respectively. The array theory is *parameterized* with the index and element theories. We can obtain multidimensional arrays by recursively defining $T_A(n)$ for n-dimensional arrays. For $n \geq 2$, we simply add $T_A(n-1)$ to the element type of $T_A(n)$.

The choice of the index and element theories will affect the expressiveness of the resulting array theory. As an instance, the index theory needs to permit existential and universal quantification in order to model properties such as "there exists an array element that is zero" or "all elements of the array are greater than zero". An example of a suitable index theory is Presburger arithmetic, i.e., linear arithmetic over integers (Chap. 5) with quantification (Chap. 9).

We start with a very general definition of the array theory. This theory is in general not decidable, however, and we therefore consider restrictions later on in order to obtain decision procedures.

7.1.1 Syntax

We define the syntax of the array theory as an extension to the combination of the index and element theories. Let $term_I$ and $term_E$ denote a term in these two theories, respectively. We begin by defining an array term $term_A$:

$$term_A : \textit{array-identifier} \mid term_A\{term_I \leftarrow term_E\} \; .$$

Next, we extend element terms to include array elements, i.e.,

$$term_E : term_A\,[\,term_I\,] \mid (previous\ rules)\ ,$$

where *previous rules* denote the grammatical rules that define $term_E$ before this extension. Finally, we extend the possible predicates in the formula by allowing equalities between array terms:

$$formula : term_A = term_A \mid (previous\ rules)\ ,$$

where here *previous rules* refer to the grammatical rules defining *formula* before this extension. The extension of the grammar with explicit equality between arrays is redundant if the index theory includes quantification, since $a_1 = a_2$ for arrays a_1 and a_2 can also be written as $\forall i.\ a_1[i] = a_2[i]$.

7.1.2 Semantics

The meaning of the new atoms and terms in the array theory is given using three axioms.

The first axiom gives the obvious meaning to the array index operator. Two array index terms have the same value if the array is the same and if the index is the same.

$$\forall a_1 \in T_A.\ \forall a_2.\ \in T_A.\ \forall i \in T_I.\ \forall j \in T_I.\ (a_1 = a_2 \wedge i = j) \implies a_1[i] = a_2[j]\ . \tag{7.2}$$

The axiom used to define the meaning of the array update operator is the **read-over-write axiom**: after the value e has been written into array a at index i, the value of this array at index i is e. The value at any index $j \neq i$ matches that in the array before the write operation at index j:

$$\forall a \in T_A.\ \forall e \in T_E.\ \forall i \in T_I.\ \forall j \in T_I.\ a\{i \leftarrow e\}[j] = \begin{cases} e & : i = j \\ a[j] & : \text{otherwise}\ . \end{cases} \tag{7.3}$$

This axiom is necessary, for example, for proving (7.1).

Finally, we give the obvious meaning to equality over arrays with the **extensionality rule**:

$$\forall a_1 \in T_A.\ \forall a_2 \in T_A.\ (\forall i \in T_I.\ a_1[i] = a_2[i]) \implies a_1 = a_2\ . \tag{7.4}$$

The array theory that includes the rule above is called the **extensional theory of arrays**.

7.2 Eliminating the Array Terms

We now present a method to translate a formula in the array theory into a formula that is a combination of the index theory and the element theory.

Aside: Array Bounds Checking in Programs
While the array theory uses arrays of unbounded size, array data structures in programs are of bounded size. If an index variable exceeds the size of an array in a program, the value returned may be undefined or a crash might occur. This situation is called an **array bounds violation**. In the case of a write operation, other data might be overwritten, which is often exploitable to gain control over a computer system from a remote location over a network. Checking that a program never violates any of its array bounds is therefore highly desirable.

Note, however, that checking array bounds in programs does not require the array theory; the question of whether an array index is within the bounds of a finite-size array requires one only to keep track of the *size* of the array, not of its contents.

As an example, consider the following program fragment, which is meant to move the elements of an array:

```
int a[N];

for(int i=0; i<N; i++)
  a[i]=a[i+1];
```

Despite the fact that the program contains an array, the verification condition for the array-bounds property does not require the array theory:

$$i < N \implies (i < N \wedge i + 1 < N) \,. \tag{7.5}$$

The translation is applicable if this combined theory includes uninterpreted functions and quantifiers over indices.

Consider Axiom (7.2), which defines the semantics of the array index operator. Now recall the definition of functional consistency, which we saw in Sect. 4.2.1. Informally, functional consistency requires that two applications of the same function must yield an equal result if their arguments are the same. It is evident that Axiom (7.2) is simply a special case of functional consistency.

We can therefore replace the array index operator by an uninterpreted function, as illustrated in the following example:

Example 7.2. Consider the following array theory formula, where a is an array with element type `char`:

$$(i = j \wedge a[j] = \texttt{'z'}) \implies a[i] = \texttt{'z'} \,. \tag{7.6}$$

The character constant `'z'` is a member of the element type. Let F_a denote the uninterpreted function introduced for the array a:

$$(i = j \wedge F_a(j) = \texttt{'z'}) \implies F_a(i) = \texttt{'z'} \,. \tag{7.7}$$

This formula can be shown to be valid with a decision procedure for equality and uninterpreted functions (Chap. 4). ▟

What about the array update operator? One way to model the array update is to replace each expression of the form $a\{i \leftarrow e\}$ by a fresh variable a' of type array. We then add two constraints that correspond directly to the two cases of the read-over-write axiom:

1. $a'[i] = e$ for the value that is written,
2. $\forall j \neq i.\ a'[j] = a[j]$ for the values that are unchanged.

This is called the **write rule**, and is an equivalence-preserving transformation on array theory formulas.

Example 7.3. The formula

$$a\{i \leftarrow e\}[i] \geq e \tag{7.8}$$

is transformed by introducing a new array identifier a' to replace $a\{i \leftarrow e\}$. Additionally, we add the constraint $a'[i] = e$, and obtain

$$a'[i] = e \implies a'[i] \geq e\ , \tag{7.9}$$

which shows the validity of (7.8). The second part of the read-over-write axiom is needed to show the validity of a formula such as

$$a[0] = 10 \implies a\{1 \leftarrow 20\}[0] = 10\ . \tag{7.10}$$

As before, the formula is transformed by replacing $a\{1 \leftarrow 20\}$ with a new identifier a' and adding the two constraints described above:

$$(a[0] = 10 \wedge a'[1] = 20 \wedge (\forall j \neq 1.\ a'[j] = a[j])) \implies a'[0] = 10\ . \tag{7.11}$$

Again as before, we transform this formula by replacing a and a' with uninterpreted-function symbols F_a and $F_{a'}$:

$$(F_a(0) = 10 \wedge F_{a'}(1) = 20 \wedge (\forall j \neq 1.\ F_{a'}(j) = F_a(j))) \implies F_{a'}(0) = 10\ .$$

▟

This simple example shows that array theory can be reduced to combinations of the index theory and uninterpreted functions, provided that the index theory offers quantifiers. The problem is that this combination is not necessarily decidable. A convenient index theory with quantifiers is Presburger arithmetic, and indeed its combination with uninterpreted functions is known to be undecidable. As mentioned above, the array theory is undecidable even if the combination of the index theory and the element theory is decidable (see Problem 7.2). We therefore need to restrict the set of formulas that we consider in order to obtain a decision procedure. This is the approach used by the reduction algorithm in the following section.

7.3 A Reduction Algorithm for a Fragment of the Array Theory

7.3.1 Array Properties

We define here a restricted class of array theory formulas in order to obtain decidability. We consider formulas that are Boolean combinations of **array properties**.

Definition 7.4 (array property). *An array theory formula is called an* array property *if and only if it is of the form*

$$\forall i_1 \ldots \forall i_k \in T_I.\ \phi_I(i_1, \ldots, i_k) \implies \phi_V(i_1, \ldots, i_k)\ , \tag{7.12}$$

ϕ_I *and satisfies the following conditions:*

ϕ_V

1. *The predicate ϕ_I, called the* index guard*, must follow the grammar*

$$\begin{aligned} \mathit{iguard} &: \mathit{iguard} \wedge \mathit{iguard} \mid \mathit{iguard} \vee \mathit{iguard} \mid \mathit{iterm} \leq \mathit{iterm} \mid \mathit{iterm} = \mathit{iterm} \\ \mathit{iterm} &: i_1 \mid \ldots \mid i_k \mid \mathit{term} \\ \mathit{term} &: \mathit{integer\text{-}constant} \mid \mathit{integer\text{-}constant} \cdot \mathit{index\text{-}identifier} \mid \mathit{term} + \mathit{term} \end{aligned}$$

   The "index-identifier" used in "term" must not be one of $i_1, \ldots, i_k$.
2. *The index variables $i_1, \ldots, i_k$ can only be used in array read expressions of the form $a[i_j]$.*

The predicate ϕ_V is called the value constraint*.*

Example 7.5. Recall Axiom (7.4), which defines the equality of two arrays a_1 and a_2 as element-wise equality. Extensionality is an array property:

$$\forall i.\ a_1[i] = a_2[i]\ . \tag{7.13}$$

The index guard is simply TRUE in this case.

Recall the array theory formula (7.1). The first and the third conjunct are obviously array properties, but recall the second conjunct,

$$\mathtt{a}' = \mathtt{a}\{i \leftarrow 0\}\ . \tag{7.14}$$

Is this an array property as well? As illustrated in Example 7.3, an array update expression can be replaced by adding two constraints. In our example, the first constraint is $a'[i] = 0$, which is obviously an array property. The second constraint is

$$\forall j \neq i.\ a'[j] = a[j]\ , \tag{7.15}$$

which does not comply with the syntax constraints for index guards as given in Definition 7.4. However, it can be rewritten as

$$\forall j.\ (j \leq i - 1 \vee i + 1 \leq j) \implies a'[j] = a[j] \tag{7.16}$$

to match the syntactic constraints. ▟

7.3.2 The Reduction Algorithm

We now describe an algorithm that accepts a formula from the array property fragment of array theory and reduces it to an equisatisfiable formula that uses the element and index theories combined with equalities and uninterpreted functions. Techniques for uninterpreted functions are given in Chap. 4.

Algorithm 7.3.1 takes an array theory formula from the array property fragment as input. Note that the transformation of array properties to NNF may turn a universal quantification over the indices into an existential quantification. The formula does not contain explicit quantifier alternations, owing to the syntactic restrictions.

As a first step, the algorithm applies the write rule (see Sect. 7.2) to remove all array update operators. The resulting formula contains quantification over indices, array reads, and subformulas from the element and index theories.

As the formula is in NNF, an existential quantification can be replaced by a new variable (which is implicitly existentially quantified). The universal quantifiers $\forall i \in T_I.\ P(i)$ are replaced by the conjunction $\bigwedge_{i \in \mathcal{I}(\phi)} P(i)$, where the set $\mathcal{I}(\phi)$ denotes the index expressions that i might possibly be equal to in the formula ϕ. This set contains the following elements:

$\boxed{\mathcal{I}(\phi)}$

1. All expressions used as an array index in ϕ that are not quantified variables.
2. All expressions used inside index guards in ϕ that are not quantified variables.
3. If ϕ contains none of the above, $\mathcal{I}(\phi)$ is $\{0\}$ in order to obtain a nonempty set of index expressions.

Finally, the array read operators are replaced by uninterpreted functions, as described in Sect. 7.2.

Example 7.6. In order to illustrate Algorithm 7.3.1, we continue the introductory example by proving the validity of (7.1):

$$\begin{array}{l} \quad (\forall x \in \mathbb{N}_0.\ x < i \implies \mathtt{a}[x] = 0) \\ \wedge\ \mathtt{a}' = \mathtt{a}\{i \leftarrow 0\} \\ \implies (\forall x \in \mathbb{N}_0.\ x \leq i \implies \mathtt{a}'[x] = 0)\ . \end{array}$$

That is, we aim to show unsatisfiability of

$$\begin{array}{l} \quad (\forall x \in \mathbb{N}_0.\ x < i \implies \mathtt{a}[x] = 0) \\ \wedge\ \mathtt{a}' = \mathtt{a}\{i \leftarrow 0\} \\ \wedge\ (\exists x \in \mathbb{N}_0.\ x \leq i \wedge \mathtt{a}'[x] \neq 0)\ . \end{array} \tag{7.17}$$

By applying the write rule, we obtain

$$\begin{array}{l} \quad (\forall x \in \mathbb{N}_0.\ x < i \implies \mathtt{a}[x] = 0) \\ \wedge\ \mathtt{a}'[i] = 0 \wedge \forall j \neq i.\ \mathtt{a}'[j] = \mathtt{a}[j] \\ \wedge\ (\exists x \in \mathbb{N}_0.\ x \leq i \wedge \mathtt{a}'[x] \neq 0)\ . \end{array} \tag{7.18}$$

Algorithm 7.3.1: ARRAY-REDUCTION

Input: An array property formula ϕ_A in NNF
Output: A formula ϕ_{UF} in the index and element theories with uninterpreted functions

1. Apply the write rule to remove all array updates from ϕ_A.
2. Replace all existential quantifications of the form $\exists i \in T_I.\ P(i)$ by $P(j)$, where j is a fresh variable.
3. Replace all universal quantifications of the form $\forall i \in T_I.\ P(i)$ by
$$\bigwedge_{i \in \mathcal{I}(\phi)} P(i) \ .$$
4. Replace the array read operators by uninterpreted functions and obtain ϕ_{UF};
5. **return** ϕ_{UF};

In the second step of Algorithm 7.3.1, we instantiate the existential quantifier with a new variable $z \in \mathbb{N}_0$:

$$\begin{aligned} &(\forall x \in \mathbb{N}_0.\ x < i \implies \mathtt{a}[x] = 0) \\ \wedge\ &\mathtt{a}'[i] = 0 \wedge \forall j \neq i.\ \mathtt{a}'[j] = \mathtt{a}[j] \\ \wedge\ &z \leq i \wedge \mathtt{a}'[z] \neq 0 \ . \end{aligned} \tag{7.19}$$

The set $\mathcal{I}$ for our example is $\{i, z\}$. We therefore replace the two universal quantifications as follows:

$$\begin{aligned} &(i < i \implies \mathtt{a}[i] = 0) \wedge (z < i \implies \mathtt{a}[z] = 0) \\ \wedge\ &\mathtt{a}'[i] = 0 \wedge (i \neq i \implies \mathtt{a}'[i] = \mathtt{a}[i]) \wedge (z \neq i \implies \mathtt{a}'[z] = \mathtt{a}[z]) \\ \wedge\ &z \leq i \wedge \mathtt{a}'[z] \neq 0 \ . \end{aligned} \tag{7.20}$$

Let us remove the trivially satisfied conjuncts to obtain

$$\begin{aligned} &(z < i \implies \mathtt{a}[z] = 0) \\ \wedge\ &\mathtt{a}'[i] = 0 \wedge (z \neq i \implies \mathtt{a}'[z] = \mathtt{a}[z]) \\ \wedge\ &z \leq i \wedge \mathtt{a}'[z] \neq 0 \ . \end{aligned} \tag{7.21}$$

We now replace the two arrays $\mathtt{a}$ and $\mathtt{a}'$ by uninterpreted functions F_a and $F_{a'}$ and obtain

$$\begin{aligned} &(z < i \implies F_a(z) = 0) \\ \wedge\ &F_{a'}(i) = 0 \wedge (z \neq i \implies F_{a'}(z) = F_a(z)) \\ \wedge\ &z \leq i \wedge F_{a'}(z) \neq 0 \ . \end{aligned} \tag{7.22}$$

By distinguishing the three cases $z < i$, $z = i$, and $z > i$, it is easy to see that this formula is unsatisfiable. ⌟

7.4 A Lazy Encoding Procedure

7.4.1 Incremental Encoding with DPLL(T)

The reduction procedure given in the previous section performs an encoding from the array theory into the underlying index and element theories. In essence, it does so by adding instances of the read-over-write rule and the extensionality rule. In practice, most of the instances that the algorithm generates are unnecessary, which increases the computational cost of the decision problem.

In this section, we discuss a procedure that generates the instances of the read-over-write (7.3) and extensionality (7.4) rules *incrementally*, which typically results in far fewer constraints. The algorithm we describe in this section follows [70] and is designed for integration into the DPLL(T) procedure (Chap. 3). It performs a lazy encoding of the array formula into equality logic with uninterpreted functions (Chap. 4). The algorithm assumes that the index theory is quantifier-free, but does permit equalities between arrays.

Preprocessing

We perform a preprocessing step before the main phase of the algorithm. The preprocessing step instantiates the first half of (7.3) exhaustively, i.e., for all expressions $a\{i \leftarrow e\}$ present in the formula, add the constraint

$$a\{i \leftarrow e\}[i] = e \; . \tag{7.23}$$

This generates a linear number of constraints. The axiom given as (7.2) is handled using the encoding into uninterpreted functions that we have explained in the previous section. The second case of (7.3) and the extensionality rule will be implemented incrementally.

Before we discuss the details of the incremental encoding we will briefly recall the basic principle of DPLL(T), as described in Chap. 3. In DPLL(T), a propositional SAT solver is used to obtain a (possibly partial) truth assignment to the theory atoms in the formula. This assignment is passed to the theory solver, which determines whether the assignment is T-consistent. The theory solver can pass additional propositional constraints back to the SAT solver in order to implement theory propagation and theory learning. These constraints are added to the clause database maintained by the SAT solver. Afterwards, the procedure reiterates, either determining that the formula is UNSAT or generating a new (possibly partial) truth assignment.

7.4.2 Lazy Instantiation of the Read-Over-Write Axiom

Algorithm 7.4.1 takes as input a set of array formula literals (array theory atoms or their negation). The conjunction of the literals is denoted by $\hat{Th}$. The algorithm returns TRUE if $\hat{Th}$ is consistent in the array theory; otherwise, it

returns a formula t that is valid in the array theory and blocks the assignment $\hat{Th}$. The formula t is initialized with TRUE, and then strengthened as the algorithm proceeds.

In line 2 the equivalence classes of the terms mentioned in $\hat{Th}$ are computed. In Sect. 4.3 we described the congruence closure algorithm for computing such classes. We denote by $t_1 \approx t_2$ the fact that terms t_1 and t_2 are in the same equivalence class.

Algorithm 7.4.1: ARRAY-ENCODING-PROCEDURE

Input: A conjunction of array literals $\hat{Th}$
Output: TRUE, or a valid array formula t that blocks $\hat{Th}$

```
1.  t := TRUE;
2.  Compute equivalence classes of terms in Th^;
3.  Construct the weak equivalence graph G from Th^;
4.  for a, b, i, j such that a[i] and b[j] are terms in Th^ do
5.      if i ≈ j then
6.          if a[i] ≉ b[j] then
7.              for each simple path p ∈ G from a to b do
8.                  if each label l on p's edges satisfies l ≉ i then
9.                      t := t ∧ ((i = j ∧ Cond_i(p)) ⟹ a[i] = b[j]);
10. return t;
```

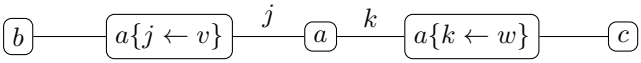
In line 3 we construct a labeled undirected graph $G(V, E)$ called the **weak equivalence graph**. The vertices V correspond to the array terms in $\hat{Th}$. The edges have an optional label, and are added as follows:

1. For each equality $a = b$ between array terms, add an unlabeled edge between a and b.
2. For each array term a and an update of that term $a\{i \leftarrow v\}$, add an edge labeled with i between their vertices.

Example 7.7. Consider the formula

$$\hat{Th} \doteq\ i \neq j \wedge i \neq k \wedge a\{j \leftarrow v\} = b \wedge a\{k \leftarrow w\} = c \wedge b[i] \neq c[i]\,. \quad (7.24)$$

The weak equivalence graph corresponding to $\hat{Th}$ is



Two arrays a and b are called *weakly equivalent* whenever there is a path from a to b in G. This means that they are equal on all array elements except,

possibly, those that are updated on the path. Arrays a, b, and c in the example above are all weakly equivalent.

Lines 4–9 generate constraints that enforce equality of array elements. This is relevant for any pair of array element terms $a[i]$ and $b[j]$ in $\hat{Th}$ where the index terms i and j are forced to be equal, according to the equivalence classes, but $a[i]$ and $b[j]$ are not. The idea is to determine whether the arrays a and b are connected by a chain of array updates where the index i is *not* used. If there is a chain with this property, then $a[i]$ must be equal to $b[j]$.

We will check whether this chain exists using our weak equivalence graph G as follows. We will consider all paths p from a to b. The path can be discarded if any of its edge labels has an index that is equal to i according to our equivalence classes. Otherwise, we have found the desired chain, and add

$$(i = j \wedge \mathit{Cond}_i(p)) \implies a[i] = b[j] \tag{7.25}$$

as a constraint to t. The expression $\mathit{Cond}_i(p)$ is a conjunction of the following constraints: $\boxed{\mathit{Cond}_i(p)}$

1. For an unlabeled edge from a to b, add the constraint $a = b$.
2. For an edge labeled with k, add the constraint $i \neq k$.

Example 7.8. Continuing Example 7.7, we have two nontrivial equivalence classes: $\{a\{j \leftarrow v\}, b\}$ and $\{a\{k \leftarrow w\}, c\}$. Hence the terms $b[i], c[i]$ satisfy $b[i] \not\approx c[i]$ and their index is trivially equal. There is one path p from b to c on the graph G, and none of its edges is labeled with an index in the same equivalence class as i, i.e., $j \not\approx i, k \not\approx i$. For this path p, we obtain

$$\mathit{Cond}_i(p) \;=\; i \neq j \wedge i \neq k \tag{7.26}$$

and subsequently update t in line 9 to

$$t := \; (i = i \wedge i \neq j \wedge i \neq k) \implies b[i] = c[i]\,. \tag{7.27}$$

Now t is added to (7.24). The left-hand side of t holds trivially, and thus, we obtain a contradiction to $b[i] \neq c[i]$. Hence, we proved that (7.24) is unsatisfiable. ∎

Note that the constraint returned by Algorithm 7.4.1 is TRUE when no chain is found. In this case, $\hat{Th}$ is satisfiable in the array theory. Otherwise t is a **blocking clause**, i.e., its propositional skeleton is inconsistent with the propositional skeleton of $\hat{Th}$. This ensures progress in the propositional part of the DPLL(T) procedure.

7.4.3 Lazy Instantiation of the Extensionality Rule

The constraints generated by Algorithm 7.4.1 are sufficient to imply the required equalities between individual array elements. In order to obtain a complete decision procedure for the extensional array theory, we need to add constraints that imply equalities between entire arrays.

Algorithm 7.4.2 is intended to be executed in addition to Algorithm 7.4.1. It generates further constraints that imply the equality of array terms.

Algorithm 7.4.2: EXTENSIONAL-ARRAY-ENCODING

Input: A conjunction of array literals $\hat{Th}$
Output: TRUE, or a valid array formula t that blocks $\hat{Th}$

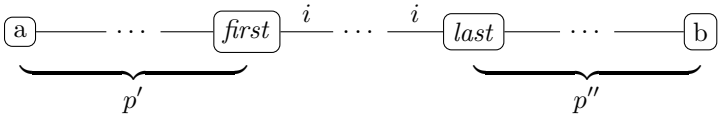
1. $t :=$ TRUE;
2. Compute equivalence classes of terms in $\hat{Th}$;
3. Construct the weak equivalence graph G from $\hat{Th}$;
4. **for** a, b such that a and b are array terms in $\hat{Th}$ **do**
5. **if** $a \not\approx b$ **then**
6. **for** each simple path $p \in G$ from a to b **do**
7. Let S be the set of edge labels of p;
8. $t := t \wedge (\bigwedge_{i \in S} Cond_i^u(p) \implies a = b)$;
9. **return** t;

An equality between two array terms is deduced as follows: Consider all pairs a, b of array terms in $\hat{Th}$ that are not equal and any chain of equalities between a and b. Choose one such chain, which we call p, and let S be the set of all distinct indices that are used in array updates in the chain. For all indices $i \in S$, do the following:

S

1. Find the array term just *before* the first edge on p labeled with i or with an index j such that $j \approx i$. Denote this term by *first*, and denote the prefix of p up to the edge with p'.
2. Find the array term just *after* the last update on p labeled with i or with an index k such that $k \approx i$. Denote this term by *last*, and denote the suffix of the path p after this edge with p''.
3. Check that $first[i]$ is equal to $last[i]$.

If this holds for all indices, then a must be equal to b. A chain of this kind in G has the following form:



Algorithm 7.4.2 checks whether such a chain exists using our graph G as follows: It considers all paths p from a to b. For each path p it computes the set S. It then adds

$$\bigwedge_{i \in S} Cond_i^u(p) \implies a = b \tag{7.28}$$

$Cond_i^u(p)$ as a constraint to t, where $Cond_i^u(p)$ is defined as follows: If there is no edge

with an index label that is equal to i in p, then

$$Cond_i^u(p) := \ Cond_i(p) \ .$$

Otherwise, it is the condition under which the updates of index i on p satisfy the constraints explained above, which is formalized as follows:

$$Cond_i^u(p) := \ Cond_i(p') \wedge \mathit{first}[i] = \mathit{last}[i] \wedge Cond_i(p'') \ . \tag{7.29}$$

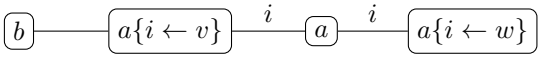
Example 7.9. Consider the following input to Algorithm 7.4.2:

$$\hat{Th} := \ v = w \quad \wedge \quad b = a\{i \leftarrow v\} \quad \wedge \quad b \neq a\{i \leftarrow w\} \ , \tag{7.30}$$

which is inconsistent. The preprocessing step (Sect. 7.4.1) is to add the instances of the first part of the read-over-write axiom (7.3). For the theory literals in $\hat{Th}$, we obtain

$$a\{i \leftarrow v\}[i] = v \quad \text{and} \quad a\{i \leftarrow w\}[i] = w \ . \tag{7.31}$$

Next, we construct the following weak equivalence graph:



Algorithm 7.4.2 will, among others, identify b and $a\{i \leftarrow w\}$ as array terms. There is one path between them, and the set S for this path is the singleton $\{i\}$. The array term *first* is $a\{i \leftarrow v\}$, and the array term *last* is $a\{i \leftarrow w\}$. Note that p' is the path from b to $a\{i \leftarrow v\}$ and that p'' is empty. We obtain

$$Cond_i^u(p) \ = \ (a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i]) \tag{7.32}$$

and subsequently add the constraint

$$a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i] \implies b = a\{i \leftarrow w\} \tag{7.33}$$

to our formula. Recall that we have added the constraints $a\{i \leftarrow v\}[i] = v$ and $a\{i \leftarrow w\}[i] = w$ and suppose that $v = w$ in all models of the formula. The decision procedure for equality logic will determine that $a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i]$ holds, and thus, DPLL(T) will deduce that $b = a\{i \leftarrow w\}$ must be true in any model of the formula, which contradicts the third literal of $\hat{Th}$ in (7.30). ⌟

7.5 Problems

Problem 7.1 (manual proofs for array logic). Show the validity of (7.1) using the read-over-write axiom.

Problem 7.2 (undecidability of array logic). A two-counter machine M consists of

- A finite set L of labels for instructions, which includes the two special labels *start* and *halt*
- An instruction for each label, which has one of the following two forms, where m and n are labels in L:
  - $c_i := c_i + 1$; **goto** m
  - **if** $c_i = 0$ **then**
    $c_i := c_i + 1$; **goto** m
    else
    $c_i := c_i - 1$; **goto** n
    endif

A configuration of M is a triple $\langle \ell, c_1, c_2 \rangle$ from $S := (L \times \mathbb{N} \times \mathbb{N})$, where ℓ is the label of the instruction that is to be executed next, and c_1 and c_2 are the current values of the two counters. The instructions permitted and their semantics vary. We will assume that $R(s, s')$ denotes a predicate that holds if M can make a transition from state s to state s'. The definition of R is straightforward. The initial state of M is $\langle \mathit{start}, 0, 0 \rangle$. We write $I(s)$ if s is the initial state. A computation of M is any sequence of states that begin in the initial state and where two adjacent states are related by R. We say that the machine *terminates* if there exists a computation that reaches a state in which the instruction has label *halt*. The problem of whether a given two-counter machine M terminates is undecidable in general.

Show that the satisfiability of an array logic formula is undecidable by performing a reduction of the termination problem for a two-counter machine to an array logic formula: given a two-counter machine M, generate an array logic formula φ that is valid if M terminates.

Problem 7.3 (quantifiers and NNF). The transformation steps 3 and 4 of Algorithm 7.3.1 rely on the fact that the formula is in NNF. Provide one example for each of these steps that shows that the step is unsound if the formula is not in NNF.

7.6 Bibliographic Notes

The read-over-write axiom (7.3) is due to John McCarthy, who used it to show the correctness of a compiler for arithmetic expressions [191]. The reads and writes correspond to loads and stores in a computer memory. Hoare and Wirth introduced the notation $(a, i : e)$ for $a\{i \leftarrow e\}$ and used it to define the meaning of assignments to array elements in the PASCAL programming language [145].

Automatic decision procedures for arrays have been found in automatic theorem provers since the very beginning. In the context of program verification, array logic is often combined with application-specific predicates, for example, to specify properties such as "the array is sorted" or to specify ranges

of indices [241]. Greg Nelson's theorem prover SIMPLIFY [101] has McCarthy's read-over-write axiom and appropriate instantiation heuristics built in.

The reduction of array logic to fragments of Presburger arithmetic with uninterpreted functions is commonplace [272, 190, 156]. While this combination is in general undecidable [105], many restrictions of Presburger arithmetic with uninterpreted functions have been shown to be decidable. Stump et al. [269] present an algorithm that first eliminates the array update expressions from the formula by identifying matching writes. The resulting formula can be decided with an EUF decision procedure (Chap. 4). Armando et al. [6] give a decision procedure for the extensional theory of arrays based on rewriting techniques and a preprocessing phase to implement extensionality.

Most modern SMT solvers implement a variant of the incremental encoding described in Sect. 7.4. Specifically, Brummayer et al. [53] used lazy introduction of functional consistency constraints in their tool BOOLECTOR, which solves combinations of arrays and bit vectors. Such a lazy procedure was used in the past also in the context of deciding arrays via quantifier elimination [97], and in the context of translation validation [229]. The definition of *weak equivalence* and the construction of the corresponding graph are given in [70].

The array property fragment that we used in this chapter was identified by Bradley, Manna, and Sipma [44]. The idea of computing "sufficiently large" sets of instantiation values is also used in other procedures. For instance, Ghilardi et al. computed such sets separately for the indices and array elements [126]. In [127], the authors sketch how to integrate the decision procedure into a state-of-the-art SMT solver. There are also many procedures for other logics with quantifiers that are based on this approach; some of these are discussed in Sect. 9.5.

7.7 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
T_I	Index type	157
T_E	Element type	157
T_A	Array type (a map from T_I to T_E)	157
$a[i]$	The element with index i of an array a	158

continued on next page

continued from previous page

Symbol	**Refers to ...**	**First used on page ...**
$a\{i \leftarrow e\}$	The array a, where the element with index i has been replaced by e	158
ϕ_I	The index guard in an array property	162
ϕ_V	The value constraint in an array property	162
$\mathcal{I}(\phi)$	Index set	163
$t_1 \approx t_2$	The terms t_1, t_2 are in the same equivalence class	166
$Cond_i(p)$	A constraint added as part of Algorithm 7.4.1	167
S	The set of indices that are used in array updates in a path	168
$Cond_i^u(p)$	A constraint added as part of Algorithm 7.4.2	168