A Solver for Arrays with Concatenation

Qinshi Wang · **Andrew W. Appel**

To appear in *Journal of Automated Reasoning,* 2023. This version dated October 2022.
This version of the article has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10817-022-09654-y.
Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

Abstract The theory of arrays has been widely investigated. But concatenation, an oper-ator that consistently appears in specifications of functional-correctness verification tools (e.g., Dafny, VeriFast, VST), is not included in most variants of the theory. Arrays with con-catenation need better solvers with theoretical guarantees. We formalize a theory of arrays with concatenation, and define the array property fragment with concatenation. Although the array property fragment without concatenation is decidable, the fragment with concate-nation is undecidable in general (e.g., when the base theory for array elements is linear integer arithmetic). But we characterize a "tangle-free" fragment; we present an algorithm that classifies verification goals in the array property fragment with concatenation as tangle-free or entangled, and that decides validity of tangle-free goals. We implement the algorithm in Coq and apply it to functional-correctness verification of C programs. The result shows our algorithm is reasonably efficient and reduces a significant amount of human effort in verification tasks. We also give an algorithm for using this array theory solver as a theory solver in SMT solvers.

Keywords array theory · program verification · decision procedure · correctness proof · proof automation

1 Introduction

We are interested in interactive automated verification of program correctness (beyond shal-low safety properties). Semi-automated verifiers, e.g. Dafny [16], VeriFast [15] and VST [3], have made great progress in this area. These verifiers provide rich specification languages

Declaration. This work was funded in part by the National Science Foundation under grant CCF-1521602 and DARPA under contract HR001120C0160. There are no conflicts of interest. Code is available open-source as described in the paper.

Qinshi Wang
Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540
E-mail: qinshiw@princeton.edu

Andrew W. Appel
Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540
E-mail: appel@princeton.edu

to describe programs' behavior. Of course, rich specification languages lead to undecidability, so these verifiers require more or less human intervention. The style of interaction varies: in Dafny and Verifast, the user keeps incrementally adding and modifying assertions, preconditions, postconditions, and loop invariants until the automatic verifier can prove the remaining straight-line code segments; in VST, the user provides similar information to the verifier in the proof script but still makes use of automatic decision procedures for subgoals.

To reduce the amount of human interaction required, we want improved decision procedures for the commonly used underlying theories. Since many programs use arrays with subscripting or lists with get-nth and update-nth, many verification tools support the theory of arrays with integer subscripting.

To describe the state of a program at some point, we often need to say the contents of array a is the *concatenation* of sequences $s_1 \cdot s_2$, for example when an array consists of two different parts, or a string is formed by concatenation. The statements in the program read from and write to the array a, so we also need *nth* and *update*. We may also have *length*, *slice* (the consecutive subsequence from index i to $j-1$), and *map* (apply function f to every element). These operators all together form our *theory of arrays with concatenation*. Even if one did not have an explicit concatenation operator, it is still often necessary to say that one part of an array satisfies one property and the other part satisfies another. The existing array solvers do not support this (see the related work paragraph later in this section).

An example of such cases is slicing arrays into pieces and operating on the pieces (as quicksort does). Figure 1 shows a few more examples that arose in the actual verification of a C array reversal function. Another example is that when reasoning about null-terminated character strings in the C language, we have the formula,

$$\text{cstring}(r,s) \quad := \quad r = s\cdot[\text{null}] \;\wedge\; \forall\, i.\; 0 \le i < |s| \implies s[i] \ne \text{null}.$$

That is, $\text{cstring}(r,s)$ says that r represents a string s; so r is a sequence of $n+1$ characters, where $|s| = n$, the last character (and only the last) is null. As another example, VigNAT [24] is a verification of network address translation (NAT) (using the VeriFast separation logic tool) whose proof of data structures heavily uses arrays (lists) and concatenation. For example, lists are used to represent the contents of arrays in the C program, and to represent sets and maps to reason about the program. Concatenation and slicing are used to merge and decompose these objects.

Indeed, concatenation can be encoded by using a universal quantifier. The concatenation $a{\cdot}b$ can be represented by a new variable c with constraints

$$|c| = |a| + |b| \;\wedge\; \forall i.\; (0 \le i < |a| \implies c[i] = a[i]) \;\wedge\; (0 \le i < |b| \implies c[i+|a|] = b[i]).$$

Although this conversion is easy and useful (our solver uses it, too), using quantifiers increases difficulty for both users and solvers. For users, using concatenation instead of quantifiers is more intuitive and concise, provides better abstraction, and will result in a computable functional model,[1] which is helpful for testing during development. That's why users of VST and VeriFast have used concatenation in functional models for program verification. For solvers, quantifiers present challenges, as we now explain.

There is no existing decision procedure to discharge proof goals involving concatenation and slicing. Bjørner *et al.*'s sequence theory [6] is just what we want, but they do not give a decision procedure. If encoded using quantifiers, Bradley *et al.*'s array property fragment [7]

[1] Concatenation is a computable function and is easy to use in a functional program that defines the behavior of the original program. Without using concatentation, it is not simple to compute the value of an array characterized by quantified formulas.

```
void reverse(int a[], int n) {
  int lo=0, hi=n, s, t;
  while (lo<hi−1)
  invariant ∃j. 0 ≤ j ≤ n − j ∧ lo = j ∧ hi = (n − j)
        ∧ a = slice(0, j, rev(a0)) · slice(j, n − j, a0) · slice(n − j, |a0|, rev(a0))
  { t = a[lo];
    s = a[hi−1];
    a[hi−1] = t;
    a[lo] = s;
    lo++; hi−−;
  }
  postcondition a = rev(a0)
}
```

$$\frac{|c| = n \quad 0 \le j \quad n-j-1 \le j \le n-j}{\text{slice}(n-n, n-(n-j), c)\cdot\text{slice}(j, n-j, c)\cdot\text{slice}(n-j, n-0, c) = c}$$

$$\frac{\begin{array}{l} n = |c| \quad 0 \le j < n-j-1 \quad |\text{rev}(c)| = |c| \\ \forall i, c.\ 0 \le i < |c| \ \rightarrow\ (\text{rev}(c))[i] = c[|c|-i-1] \end{array}}{\begin{array}{l} \text{update}\Big(j, \\ \quad \text{update}\big(n-j-1, \\ \qquad \text{slice}(0, j, \text{rev}(c))\cdot\text{slice}(j, n-j, c)\cdot\text{slice}(n-j, n, \text{rev}(c)), \\ \qquad \text{slice}(0, j, \text{rev}(c))\cdot\text{slice}(j, n-j, c)\cdot\text{slice}(n-j, n, \text{rev}(c))[j]\big), \\ \quad \text{slice}(0, j, \text{rev}(c))\cdot\text{slice}(j, n-j, c)\cdot\text{slice}(n-j, n, \text{rev}(c))[n-j-1]\Big) \\ = \text{slice}(0, j+1, \text{rev}(c))\cdot\text{slice}(j+1, n-(j+1), c) \\ \qquad \cdot\text{slice}(n-(j+1), n, \text{rev}(c)) \end{array}}$$

$$\frac{|c| = n \quad 0 <= j < n-j-1 \quad |\text{rev}(c)| = |c|}{c[n-j-1] = \text{slice}(0, j, \text{rev}(c))\cdot\text{slice}(j, n-j, c)\cdot\text{slice}(n-j, n, \text{rev}(c))[n-j-1]}$$

$$\frac{0 \le j < |c|-j-1 \qquad |b| = |c|}{\begin{array}{l} \text{update}(j, \text{update}(|c|-j-1, b, b[j]), c[|c|-j-1]) \\ = \text{slice}(0, j, b)\cdot\text{slice}(|c|-j-1, |c|-j, c)\cdot\text{slice}(j+1, |c|, \text{slice}(0, |c|-j-1, b)\cdot\text{slice}(|c|-j, |c|, b)) \end{array}}$$

Fig. 1: C program and some of the array-theory problems that arise in its proof.
Our solver handles all of these, but in the second example (since it has no primitive knowledge of the **rev** function or its properties), the user must "manually" apply the quantified formula shown just above the line.

does not allow index shifting (expressions of the form $a[i+n]$), which is necessary to encode concatenation and slicing. To fill this gap, we describe a property called *entanglement*. Our algorithm classifies proof goals into tangle-free or entangled and (if tangle-free) decides their validity. In verifying imperative programs, we find informally that most proof goals are tangle-free[2]; and on entangled goals our solver fails quickly and shows the reason of

[2] Not only in our own program verifications: We examined the VigNAT [24] proofs (github.com/vignat) in files listexex.gh and map–impl.c. Of the 73 lemmas in listexex.gh, 50 are expressible in our theory. The rest of lemmas involve operators unsupported in our theory, e.g. filter and elements-are-distinct. Of these, 37 lemmas are quantifier-free (hence tangle-free) and 13 lemmas are quantified and tangle-free; none are entangled. (We determined this by inspection—none have the kind of recurrence that would lead to entanglement.) All 50 lemmas would be solved by our solver. In fact, if VeriFast had been equipped with our solver, none of these 50 lemmas in listexex.gh would be needed. The file map–impl.c has 110 lines of C code and 2500 lines of specification and proof. Within the proof there are approximately 100 applications of list-operation lemmas (such as the ones in listexex.gh, listex.gh, and list.gh) which could be fully or partially automated

entanglement to the user as a hint to perform some manual proof steps until the solver can solve the goal.

Our algorithm exploits the idea that concatenation can be encoded by universally quantified elementwise equations. Intuitively, a proof goal with quantification is hard to solve if its quantified assumptions encode recurrences. For example, consider a proof goal of the form

$$(\forall i:\mathbb{Z}.\ 0 \le i < |a|-1 \implies P(a[i], a[i+1])) \implies Q(a[0], a[|a|-1]). \tag{1}$$

This is hard to decide, because the relationship between $a[0]$ and $a[|a|-1]$ can be intricate, even if P and Q can be expressed in a decidable theory.

Remark 1 There exists a decidable base theory for which the array property fragment with concatenation is undecidable.

Proof Let the base theory (for array elements) be linear integer arithmetic with tuples of integers. The halting problem for two-counter machines (which is undecidable [20, Theorem 14.1-1]) can be encoded in such a proof goal. The elements of a are tuples of three integers each, corresponding to the values of the two counters and the program counter. The transition relation between states, namely P, can be expressed in the quantifier-free linear integer arithmetic theory. Let Q encode that $a[0]$ is the starting state and $a[|a|-1]$ is not a halting state. Then the proof goal is valid if and only if the two-counter machine does not halt, so it is undecidable.

Another example that may arise in string matching is assumptions of the form $a \cdot b_1 = b_2 \cdot a$. If $|a| > |b_2|$, then some suffix of a equals a prefix of a. In other words, for some integer n,

$$\forall i:\mathbb{Z}.\ 0 \le i < |a|-n \implies a[i] = a[i+n]. \tag{2}$$

That implies a has a periodic structure $a = a_0a_0\ldots a_0a_1$, where a_0 is the periodic part and a_1 is a prefix of a_0. This periodic structure is very hard to analyze.

Actually, the examples in (1) and (2) share the same difficulty, which is relating $a[i]$ and $a[i+1]$ ($a[i]$ and $a[i+n]$, respectively) in a quantified formula. We call this phenomenon index shifting. We apply a test to detect index shifting in proof goals. A proof goal is tangle-free if it does not have index shifting; otherwise it is entangled. And then we show tangle-free proof goals can be decided by our algorithm.

Contributions. (1) We describe a decision procedure for a useful fragment of arrays/lists/-sequences with concatenation and indexing. (2) The algorithm uses quantifiers to represent equations, which allows simple manipulation of equations together with other formulas quantified on indices. (3) It improves on the fragment of Bradley et al. [7] by allowing non-constant index offsets, which can express general concatenation. It improves on the semidecision procedure by Ge and de Moura [12], by giving a clear condition of when it can decide validity. (4) We implement our algorithm in Coq and Ltac, and integrate with VST. The implementation not only solves proof goals but also generates proof terms in Coq. (5) We demonstrate that the solver substantially reduces human effort in verification of imperative programs.

using our solver—which would significantly reduce the number of lines of human-written proof-script. In all of these subgoals we did not identify any that have the kind of recurrence that would lead to entanglement.

Related work. Theories of arrays, lists, or sequences have been intensively researched. Many works treat arrays as functions mapping integers to elements. McCarthy [19] gave a decision procedure for the quantifier-free fragment of the array theory with read and write. Stump et al. [23] gave a decision procedure for arrays with extensionality. Bradley et al. [7] proposed the array property fragment, a decidable fragment that allows universal quantifiers on integer variables with certain restrictions. This fragment is expressive enough to encode some useful properties such as sortedness. Ge and de Moura [12] further generalized this fragment to allow index offset; but where our algorithm detects and informs the user when a goal is outside the fragment, theirs does not distinguish between "outside the fragment" and "invalid" (and in either of those cases may not terminate). Ge and de Moura's fragment can encode concatenation by index offset; Bradley et al.'s cannot (no index offset).

Some procedures are specialized to integer arrays. Habermehl et al. [13] gave a method for the satisfiability of integer arrays with constraints of the form

$$\forall i.\ l \leq i < r \implies a[i] \leq a[i+c] + d,$$

where c and d are integer literals, by encoding the formulas into counter automata. Daca et al. [10] generalized this method to decide the array folds theory, an integer array theory with a fold operator. *Fold* allows the accumulator to contain a state from a finite set of states and some integer counters. On each step, the folding function determines an operation based on the state and the current array element, and the operation includes the transition of state and the increment/decrement of the counter by constants. These two pieces of work only support constant index offsets, so they do not support concatenation in general.

Another track of research is the theory of sequences with concatenation, which can be regarded as a free monoid. Makanin [18] gave a decision procedure for quantifier-free equations in a free monoid, which was improved by Plandowski [21,22]. Neither of these theories supports the *length* operator, which is crucial to express indexing, update, and slicing.

The theory of strings has attracted recent interest because of applications in web safety problems, such as protection from cross-site scripting (XSS) attacks and SQL injections [9, 14]. Most string theories support word equations (between concatenations of string variables and constants), length constraints, and membership in regular languages. Some theories [14] also support regular transducers. String theories have a much simpler base type (finite alphabets without arithmetic), and it remains unclear whether the satisfiability of the string theory with word equations and length constraints is decidable [11]. Some decidable fragments have been investigated, such as straight-line fragment [14,17], acyclic fragment [1], chain-free fragment [2], and nonoverlapping condition [25]. These fragments all capture the same difficulty as this paper, namely direct or indirect equations between different parts of the same array/string variable; see the further discussion in Section 5. But string theories do not permit the element sort to be a sort from some other theory with cooperating decision procedures, so strings are not the best abstraction for our applications in program verification.

2 Theory of arrays with concatenation

We describe the theory in terms of the standard many-sorted first-order logic for satisfiability modulo theories (SMT). For detailed formalization of the logic, we refer the reader to Barrett and Tinelli's book chapter [4]. In particular, a theory is a class of interpretations. Consider a base theory T_B on base signature Σ_B and sort set $\mathcal{S}_\mathsf{B}$. For simplicity, we assume that T_B has only one interpretation, denoted by I_B, but the correctness of our algorithm is not hard to

Family	Type signature	Standard Interpretation
length_S	$A(S) \to \mathbb{Z}$	$\text{length}_S([x_0,x_1,\ldots,x_{n-1}]) = n$
nth_S	$\mathbb{Z} \times A(S) \to S$	$\text{nth}_S(i,[x_0,x_1,\ldots,x_{n-1}]) = \begin{cases} x_i, & \text{if } 0 \le i < n \\ d_S, & \text{otherwise} \end{cases}$
repeat_S	$S \times \mathbb{Z} \to A(S)$	$\text{repeat}_S(x,n) = \underbrace{[x,x,\ldots,x]}_{\max(0,n) \text{ times}}$
app_S	$A(S) \times A(S) \to A(S)$	$\text{app}_S([x_0,x_1,\ldots,x_{n-1}],[y_0,y_1,\ldots,y_{m-1}])$ $= [x_0,x_1,\ldots,x_{n-1},y_0,y_1,\ldots,y_{m-1}]$
slice_S	$\mathbb{Z} \times \mathbb{Z} \times A(S) \to A(S)$	$\text{slice}_S(i,j,[x_0,x_1,\ldots,x_{n-1}])$ $= \begin{cases} [x_l,\ldots,x_{r-1}], & \text{if } l < r, \\ [\,], & \text{otherwise,} \end{cases}$ where $l = \max(i,0), r = \min(j,n)$
map_f	$A(S_1) \times \cdots \times A(S_k) \to A(S')$ for $f : S_1 \times \cdots \times S_k \to S'$	$\text{map}_f([x_{1,0},\ldots,x_{1,n_1-1}],\ldots,[x_{k,0},\ldots,x_{k,n_k-1}])$ $= [f(x_{1,0},\ldots,x_{k,0}),\ldots,$ $f(x_{1,n_{\min}-1},\ldots,x_{k,n_{\min}-1})]$ where $n_{\min} = \min(n_1,\ldots,n_k)$

Fig. 2: Array operators

generalize to the case with multiple interpretations, since it is proved for any single interpretation. We assume I_{B} interprets the sort $\mathbb{Z}$ and arithmetic operators as the standard integers and arithmetic. Assume there is a base solver $\mathcal{O}_{\mathsf{B}}$ to decide the T_{B}-validity of quantifier-free formulas with uninterpreted functions, which means whether a formula is satisfied by every interpretation that extends I_{B} on uninterpreted functions and free variables. Without loss of generality, we assume $(\min,\max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}) \in \Sigma_{\mathsf{B}}$ and I_{B} gives them standard interpretations. We write $\min(n_1,\ldots,n_{k-1},n_k)$ for $\min(n_1,\ldots\min(n_{k-1},n_k)\ldots)$, and the same for max. We assume each sort $S \in \mathcal{S}_{\mathsf{B}}$ has at least one element and there is a special variable d_S in the sort S serving as the "default" value of S. The default value is necessary in order to define the nth-element function as a total function, as it can yield d_S when the index is out of bounds.

The theory of arrays with concatenation has an array sort $A(S)$ for each base sort $S \in \mathcal{S}_{\mathsf{B}}$, in addition to $\mathcal{S}_{\mathsf{B}}$. Its universe consists of finite length sequences of S: $U_{A(S)} = \bigcup_{n=0}^{\infty}(U_S^{I_{\mathsf{B}}})^n$. There are six array operators:

$\text{length}_S(a)$ returns the length of an array (notation $|a|$);
$\text{nth}_S(i,a)$ returns the i-th (0-indexed) element (notation $a[i]$);
$\text{repeat}_S(x,n)$ generates an array by repeating x for n times (notation x^n);
$\text{app}_S(a,b)$ concatenates two arrays into one array (notation $a{\cdot}b$);
$\text{slice}_S(i,j,A)$ takes a slice of an array from position i to position j;
$\text{map}_f(\cdot)$ applies f elementwise to corresponding elements of k arrays.

Fig. 2 defines their behavior, including exceptional cases. The update operator is not included here, because it can be defined using other operators as

$$\text{update}(i,a,x) \;=\; \text{if } 0 \le i < |a| \text{ then } \text{slice}(0,i,a){\cdot}x^1{\cdot}\text{slice}(i+1,|a|,a) \text{ else } a.$$

Equality between arrays will be interpreted by extensionality, i.e. we interpret $a =_{A(S)} b$ as the formula

$$|a| =_{\mathbb{Z}} |b| \;\wedge\; \forall i.\ 0 \le i < |a| \implies a[i] =_S b[i]. \qquad (3)$$

The quantifier-free fragment permits such array equations only in positive positions, so that the $\forall i$ can be removed and i can be replaced with a fresh free variable.

The array property fragment with concatenation (APFC) allows *quantified formulas* of the form

$$\forall \vec{\mathbf{i}}.\ \varphi_I(\vec{\mathbf{i}}) \implies \varphi_V(\vec{\mathbf{i}}),$$

where $\vec{\mathbf{i}}$ is a vector of integer variables, φ_I is the index guard, and φ_V is the value constraint. The index guard is written in the grammar,

$$\begin{aligned} \text{iguard} &:= \text{iguard} \wedge \text{iguard} \mid \text{iguard} \vee \text{iguard} \mid \text{atom} \\ \text{atom} &:= \text{expr} \leq \text{expr} \\ \text{expr} &:= \text{uvar} + \text{pexpr} \mid \text{pexpr} \\ \text{uvar} &:= (\text{any variable from } \vec{\mathbf{i}}) \\ \text{pexpr} &:= (\text{any term without variables from } \vec{\mathbf{i}}). \end{aligned}$$

The *value constraint* is a quantifier-free formula of sort $\mathbb{P}$ (*i.e.,* Proposition) in which

- array subscripts (arguments of nth) take only the form [uvar + pexpr] or [uvar] or [pexpr];
- quantified variables uvar appear only in array subscripts; and
- arguments of map, repeat, and slice are pexprs—that is, contain no variables from $\vec{\mathbf{i}}$ (not even in subscripts).

Proof goals in the APFC are Boolean combinations of quantified formulas and quantifier-free formulas using conjunction, disjunction, and negation (and implication). Let P be a predicate on an array element. A simple example of a proof goal in the APFC is

$$(\forall i.\ 0 \leq i < |a{\cdot}b| \implies P((a{\cdot}b)[i])) \implies (\forall i.\ 0 \leq i < |b{\cdot}a| \implies P((b{\cdot}a)[i])). \quad (4)$$

That is, if each element of $a{\cdot}b$ satisfies property P, then each element of $b{\cdot}a$ satisfies P.

Throughout this paper, we describe the problem as deciding validity of formulas since it is our main interest. To decide satisfiability, one can reduce it to validity.

Comparison with the array property fragment. Comparing with the array property fragment of Bradley et al. [7], the APFC allows array indices of the form $i+n$, and comparison between $i+n$ and $j+m$ in index guards. This makes it possible to express concatenation but also raises a challenge. We will address the challenge by classifying formulas into tangle-free and entangled ones and only deciding validity of tangle-free formulas.

3 Classification and decision procedure

We describe the algorithm as a procedure that reduces the original formula into an equivalid formula in the base theory, and decide its validity using the base solver $\mathcal{O}_{\mathsf{B}}$. We assume that no uninterpreted functions have array arguments or array results. (In §9, we discuss the extension with uninterpreted functions and combining with other theories through the Nelson-Oppen procedure.)

The classification and decision procedure for goals in the APFC consists of six phases.

1. *Preprocessing:* Substitute with definitional equations, unfold shorthands and Skolemize quantified variables in positive positions.

2. *Reducing subscripted expressions whose subscript-terms have no quantified variables* until all such subscripted terms are simply (free) variables.
3. *Reducing array-subscript terms with quantified variables* in quantified formulas until *all* subscripted terms are variables; denote the result with ψ_1.
4. *Classification:* Construct the *index propagation graph* and the test formula Δ whose validity indicates whether the goal is tangle-free or entangled, and invoke the base solver $\mathcal{O}_\mathrm{B}$ to decide it. If it is entangled, report that the goal is entangled with the reason of entanglement and then terminate.
5. *Instantiation:* Instantiate the quantifiers by a set calculated from the array indexing terms in ψ_1 and the index propagation graph, resulting in ψ_2.
6. *Decision:* Convert ψ_2 into a formula ψ_B in the base theory T_B; invoke the base solver $\mathcal{O}_\mathrm{B}$ to decide validity of ψ_B, which implies the validity of ψ_2 and (therefore) ψ.

The intuition of the algorithm is that it first performs reductions to remove app, repeat, slice, and map. Then it needs to handle quantified formulas. If it sees a term $a[n]$ and a quantified formula $\forall i : \mathbb{Z}.\ \varphi(a[i+m])$, it instantiates i with $n-m$, so that $a[n]$ is the same as the instantiated $a[i+m]$. This instantiation may not terminate—so *before instantiation* we need to analyze whether it terminates, which is the classification phase. If it terminates, we say the goal is tangle-free. Then we can show that the instantiation is not only sound but also complete. So the algorithm can decide validity.

We explain this algorithm in detail in the rest of this section. We will use the following running example, which is slightly modified from (4)—if every element of $a{\cdot}b$ satisifies P, then so does every element of $b{\cdot}a$:

Example 1 Let a,b,c,d be array variables (of a certain base sort), and P be a predicate on an array. The example goal is

$$(c = a{\cdot}b \wedge d = b{\cdot}a \wedge \forall i.\ 0 \le i < |c| \implies P(c[i])) \implies \forall i.\ 0 \le i < |d| \implies P(d[i]). \quad (5)$$

3.1 Preprocessing

The first step is to substitute with definitional equations, which are array equations that appear as global assumptions and one of whose sides is an array variable. A formula φ is a global assumption in a goal ψ if ψ can be converted to the form $\varphi \implies \psi'$ by Boolean conversions. For example, in (5), $c = a{\cdot}b$ and $d = b{\cdot}a$ are definitional equations. We can substitute c and d and the result is

$$(\forall i.\ 0 \le i < |a{\cdot}b| \implies P((a{\cdot}b)[i])) \implies \forall i.\ 0 \le i < |b{\cdot}a| \implies P((b{\cdot}a)[i]).$$

The second step is to unfold shorthands. For example, if there are array equations, we can use (3) to convert to quantified formulas.

The third step is to Skolemize quantified variables in positive positions (i.e. remove universal quantifiers that appear in positive positions, and rename the variables with fresh names, so they become free variables, which are implicitly quantified at the top level). Preprocessing our running example, the second quantified formula is at positive position, so we rename i to k and make it implicitly quantified at the top level. The result is

$$(\forall i.\ 0 \le i < |a{\cdot}b| \implies P((a{\cdot}b)[i])) \implies 0 \le k < |b{\cdot}a| \implies P((b{\cdot}a)[k]).$$

$$\text{LENGTH_REPEAT}\ \frac{\psi(\max(0,n))}{\psi(|x^n|)} \qquad \text{LENGTH_APP}\ \frac{\psi(|a|+|b|)}{\psi(|a{\cdot}b|)}$$

$$\text{LENGTH_SLICE}\ \frac{\begin{array}{c}\max(i,0) < \min(j,|a|) \implies \psi(\min(j,|a|)-\max(i,0)) \\ \max(i,0) \geq \min(j,|a|) \implies \psi(0)\end{array}}{\psi(|\mathrm{slice}(i,j,a)|)}$$

$$\text{LENGTH_MAP}\ \frac{\psi(\min(|a_1|,\dots,|a_k|))}{\psi(\left|\mathrm{map}_f(a_1,\dots,a_k)\right|)}$$

$$\text{NTH_REPEAT}\ \frac{0 \leq i < n \implies \psi(x) \qquad \neg(0 \leq i < n) \implies \psi(d_S)}{\psi(x^n[i])}$$

$$\text{NTH_APP}\ \frac{\begin{array}{c}0 \leq i < |a| \implies \psi(a[i]) \\ |a| \leq i < |a|+|b| \implies \psi(b[i-|a|])) \qquad \neg(0 \leq i < |a|+|b|) \implies \psi(d_S)\end{array}}{\psi((a{\cdot}b)[i])}$$

$$\text{NTH_SLICE}\ \frac{\begin{array}{c}\mathrm{cond}(a,i,j,k) \implies \psi(a[i+\max(j,0)]) \\ \neg\mathrm{cond}(a,i,j,k) \implies \psi(d_S) \qquad \text{where } \mathrm{cond}(a,i,j,k) := 0 \leq i \wedge i+\max(j,0) < \min(k,|a|)\end{array}}{\psi(\mathrm{slice}(j,k,a)[i])}$$

$$\text{NTH_MAP}\ \frac{\begin{array}{c}0 \leq i < \min(|a_1|,\dots,|a_k|) \implies \psi(f(a_1[i],\dots,a_k[i])) \\ \neg(0 \leq i < \min(|a_1|,\dots,|a_k|)) \implies \psi(d_{S'})\end{array}}{\psi(\mathrm{map}_f(a_1,\dots,a_k)[i])}$$

Fig. 3: Reduction rules for array expressions without quantified variables. Proof goals matching the bottom of a rule will be replaced by the top of the rule.

3.2 Reducing subscripted expressions without quantified variables

We simplify terms without quantified variables so that the only array-sorted terms are (free) variables. Observe that no predicates have arguments of array sorts, so arrays can only be subterms of these arguments. If any subterm has array sort, there is at least one highest-level subterm of array sort, whose parent term has a base sort. The only functions with array arguments and nonarray results are *length* and *nth*. Each array argument is either a variable or a functional term led by *repeat*, *app*, *slice* or *map*. So we can reduce the number of distinct array subterms by simplifying subterms of these 2×4 possible forms. This is what the rules in Fig. 3 do: match the bottom of the rule to the proof goal, yielding new proof goals as the top of the rule. Then the running example becomes three subgoals

$$\begin{array}{l}
0 \leq k < |b| \implies \\
\qquad (\forall i.\ 0 \leq i < |a|+|b| \implies P((a{\cdot}b)[i])) \implies 0 \leq k < |b|+|a| \implies P(b[k]) \\
|b| \leq k < |b|+|a| \implies \\
\qquad (\forall i.\ 0 \leq i < |a|+|b| \implies P((a{\cdot}b)[i])) \implies 0 \leq k < |b|+|a| \implies P(a[k-|b|]) \\
k < 0 \vee k \geq |b|+|a| \implies \\
\qquad (\forall i.\ 0 \leq i < |a|+|b| \implies P((a{\cdot}b)[i])) \implies 0 \leq k < |b|+|a| \implies P(d_S).
\end{array}$$

Notice that the last case can be solved immediately because the assumptions on the range of k are contradictory.

$$\forall\text{NTH_REPEAT}\ \frac{\psi\begin{pmatrix}\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge 0\le i<n \implies F(x)\right)\\ \wedge\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge\neg(0\le i<n) \implies F(d_{S_x})\right)\end{pmatrix}}{\psi(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies F((x^n)[i]))}$$

$$\forall\text{NTH_APP}\ \frac{\psi\begin{pmatrix}\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge 0\le i<|a| \implies F(a[i])\right)\\ \wedge\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge |a|\le i<|a|+|b| \implies F(b[i-|a|])\right)\\ \wedge\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge\neg(0\le i<|a|+|b|) \implies F(d_S)\right)\end{pmatrix}}{\psi(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies F((a\cdot b)[i])}$$

$$\forall\text{NTH_SLICE}\ \frac{\psi\begin{pmatrix}\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ (\varphi_I(\vec{\mathbf{i}})\wedge \mathrm{cond}(a,i,j,k)) \implies F(a[i+\max(j,0)])\right)\\ \wedge\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ (\varphi_I(\vec{\mathbf{i}})\wedge\neg\mathrm{cond}(a,i,j,k)) \implies F(d_S)\right)\end{pmatrix}\\ \text{where } \mathrm{cond}(a,i,j,k) := 0\le i\wedge i+\max(j,0)<\min(k,|a|)}{\psi(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies F(\mathrm{slice}(j,k,a)[i]))}$$

$$\forall\text{NTH_MAP}\ \frac{\psi\begin{pmatrix}\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge 0\le i<\min(|a_1|,\ldots,|a_k|) \implies F(f(a[i],\ldots,a_k[i]))\right)\\ \wedge\left(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}})\wedge\neg(0\le i<\min(|a_1|,\ldots,|a_k|)) \implies F(d_S)\right)\end{pmatrix}}{\psi(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies F(\mathrm{map}_f(a_1,\ldots,a_k)[i])}$$

$$\forall\text{PRUNE}\ \frac{\psi(\top)\qquad \neg\psi(\top) \implies \forall\vec{\mathbf{i}}:\mathbb{Z}.\ \neg\varphi_I(\vec{\mathbf{i}})}{\psi(\forall\vec{\mathbf{i}}:\mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies \varphi_V(\vec{\mathbf{i}}))}$$

Fig. 4: Rewrite rules for quantified formulas. F may have free variables from $\vec{\mathbf{i}}$, as it is a value constraint $\varphi_V(\vec{\mathbf{i}})$ with a hole.

All these rewrite rules are sound and do not turn a valid goal into an invalid goal. That is, we have proved in Coq from the interpretation of the array theory (Fig. 2) that the formulas before and after rewriting are equivalent.

3.3 Reducing subscripted expressions with quantified variables

The next phase puts the goal into the APFC without app, repeat, slice, or map. The rewrite rules for quantified formulas are displayed in Fig. 4. The first four rules simplify array terms under quantifiers. These rules concern one quantified variable at a time, so we use i to denote the variable concerned and use $\vec{\mathbf{i}}$ to denote all variables in the quantified formula, including i. These rewrite rules are sound and complete (do not turn provable goals into unprovable goals) whether the formula appears in a positive or a negative position, but the algorithm only uses them in negative positions. Each rewrite rule converts a quantified formula into a conjunction of quantified formulas. Given a formula in the APFC, the formula produced by the rewriting procedure is still in the APFC, and only has array length and indexing terms of the forms $|a|$ and $a[i+n]$.

The last rewrite rule ∀PRUNE removes quantified formulas whose index guard clauses are unsatisfiable. This avoids unnecessary instantiation. In §3.4, we will see that this prun-

ing removes edges from the index propagation graph and allows classifying more goals as tangle-free. When attempting to apply this rule, the second premise is decided using a conservative approximation that all the quantified formulas are replaced by $\top$. Then its validity is decided using $\mathcal{O}_{\mathrm{B}}$ (described in detail later in §3.6). The rule is applied only when the second premise is decided valid.

After reducing array expressions inside quantifiers, the goal will be in the APFC without any app, repeat, slice, or map. The running example's first subgoal is reduced to,

$$\begin{aligned}&0 \le k < |b| \implies \\ &\qquad (\forall i.\ 0 \le i < |a| \implies P(a[i])) \implies \\ &\qquad (\forall i.\ |a| \le i < |a|+|b| \implies P(b[i-|a|])) \implies 0 \le k < |b|+|a| \implies P(b[k]),\end{aligned}$$

by applying ∀NTH_APP and applying ∀PRUNE on the out-of-bounds branch.

3.4 Classification

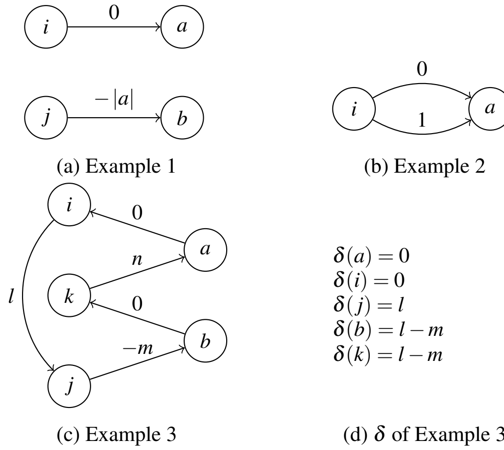
At this stage, we need to consider instantiating the quantifiers. We first rename the universally quantified variables into distinct names, in order to handle them clearly. So the running example's first subgoal is turned into

$$\begin{aligned}&0 \le k < |b| \implies \\ &\qquad (\forall i.\ 0 \le i < |a| \implies P(a[i])) \implies \\ &\qquad (\forall j.\ |a| \le j < |a|+|b| \implies P(b[j-|a|])) \implies 0 \le k < |b|+|a| \implies P(b[k]).\end{aligned}$$

Let us imagine a "straw-man" naive instantiation algorithm: Intuitively, since we have $b[k]$, we should instantiate j with $k+|a|$ because then $b[j-|a|]$ is just $b[k]$. Generally, for a universally quantified variable i that is used as $a[i+n]$ in the value constraint, we should instantiate i with $x-n$ when we have a term $a[x]$. And having instantiated i with y, we will have a term $a[y+n]$, which may trigger more instantiation. Also, when there is an index guard atom of the form $i+n \le j+m$ and i is instantiated with x, then we have $x+n \le j+m$, i.e. $x+n-m \le j$, which is a hint to instantiate j with $x+n-m$.

Such instantiation may cause instantiation loops. So we introduce the *index propagation graph* (IPG) as a way to analyze instantiation and detect loops. The nodes of the IPG are array variables and quantified variables, and each edge is labeled with an integer term (also called weight). To instantiate completely, for a term $a[x]$, the algorithm should traverse the graph from node a, propagate x through the edges, add x with the labels (e.g. it becomes $x+n$ after propagating through as edge with label n), and instantiate the quantified variables with the resulted indices. If the IPG has a cycle that produces $a[x+n]$ when instantiating $a[x]$, then (unless $n=0$) instantiation would go into an infinite loop. Here n is actually the accumulated weight (the sum of the labels) of the cycle. If the IPG has a cycle with nonzero accumulated weight, that signals *entanglement*; with no cycles, or only zero-weighted cycles, *tangle-free*. We summarize the construction of the IPG formally.

Definition 1 (Index Propagation Graph) For formula ψ such that array subterms are variables and universally quantified variables have distinct names, its index propagation graph (IPG) is a directed multigraph (V,E) with an integer term on each edge. The node set V consists of array variables and universal variables. We use (u,v,w) to denote an edge from u to v with label w. For each subterm of the form $a[i+n]$ in each value constraint φ_V, add



(a) Example 1 (b) Example 2 (c) Example 3 (d) δ of Example 3

Fig. 5: Index propagation graphs for examples. For clarity, we only draw one of each pair of edges in the IPG, so each displayed edge implies another edge with an opposite label in the opposite direction.

edges (i,a,n) and $(a,i,-n)$. If there is a comparison in the form of $i+n \leq j+m$ in the guard formula φ_I, add edges $(i,j,n-m)$ and $(j,i,m-n)$. Edges in an IPG are in pairs, such that each pair of edges connects the same pair of nodes in opposite directions with opposite labels.

Besides the running example, we also demonstrate the IPG using the following two examples.

Example 2 For array variable a and integer variable j,

$$(\forall i : \mathbb{Z}.\ 0 \leq i < |a|-1 \implies a[i] = a[i+1]) \implies 0 \leq j < |a| \implies a[0] = a[j].$$

Example 3 For array variables a and b and integer variables l,m,n, consider a proof goal with two quantified assumptions as follows:

$$\forall k : \mathbb{Z}.\ 0 \leq k < |b| \implies a[k+n] = b[k],$$
$$\forall ij : \mathbb{Z}.\ 0 \leq i < |a| \wedge 0 \leq j-m < |b| \wedge i \leq j-l \implies a[i] \leq b[j-m].$$

The IPGs of these examples are as shown in Fig. 5. Example 1 has no cycles. Example 2 has a cycle, which has accumulated weight 1. So the cycle has nonzero weight and the goal is entangled (that is, Example 2 has a recursive restriction). Example 3 also has a cycle, whose accumulated weight is $l+n-m$. So the algorithm will first decide whether $l+n-m=0$ is provable to determine whether the goal is tangle-free.

Checking the IPG for nonzero cycles. We now describe a graph algorithm for detecting nonzero cycles in the IPG. It is unnecessary to enumerate all the (exponentially many) cycles. Instead, it is enough to choose a spanning forest F of the IPG (a spanning tree from each component), ignoring edge direction, and then construct a function $\delta : V \to \{\text{integer terms}\}$ as follows. For each connected component C of the IPG, arbitrarily choose a reference node

u_C, and set $\delta(u_C) = 0$. We then define δ for the remaining nodes by depth/breadth first search through F such that

$$\delta(u) + w = \delta(v), \tag{6}$$

for every directed edge $(u,v,w) \in F$. The construction of δ for Example 3 is shown in Fig. 5. Then we try to prove that (6) also holds for edges not in F. If we can prove that (6) holds for edges not in F from the assumptions, then (6) holds for every edge, and the IPG's cycles are all zero-weighted, because for every cycle C, we have the following formula by applying (6) on each edge of C and taking the sum:

$$\sum_{(u,v,w)\in C} \delta(u) + \sum_{(u,v,w)\in C} w = \sum_{(u,v,w)\in C} \delta(u), \tag{7}$$

and then $\sum_{(u,v,w)\in C} w = 0$.

Conversely, if we can prove that the IPG's cycles are all zero-weighted from the assumptions, we can prove (6) for each off-forest edge. That is because there is a cycle C for each off-forest edge e, that C contains only e and some on-forest edges. Then $\sum_{(u,v,w)\in C} w = 0$ implies (7), which implies (6) since we have (6) for all on-forest edges.

Proving equations in the form of (6) might need assumptions from ψ_1 (e.g. Example 3). In other words, we need to prove $\neg\psi_1 \implies$ (6). Then we have the same challenge of instantiation as in the original goal. So we construct an approximated version of ψ_1, denoted by ψ_1', by overapproximating quantified formulas with $\top$, i.e. dropping quantified assumptions. Then define the test formula

$$\Delta \quad := \quad \neg\psi_1' \implies \bigwedge_{(u,v,w)\in E\setminus F} \delta(u) + w = \delta(v).$$

We then check the validity of Δ by invoking the base solver as we will describe in §3.6. Note that the validity of Δ is independent of the choice of F and reference nodes, because it only depends on all cycles in the IPG. The goal is classified as tangle-free if Δ is proved valid, and otherwise entangled.[3] If Δ cannot be proved, the algorithm identifies the equation that it failed to prove and stops.[4] The user may prove the equation manually and put it into the assumptions, so the algorithm can classify the goal as tangle-free.

Example 2 is entangled because we cannot prove $1 = 0$. For Example 3, we need to prove (6) for the off-forest edge (k,a,n), i.e. $l - m + n = 0$. The goal is tangle-free if $l - m + n = 0$ is provable from $\neg\psi_1'$.

3.5 Instantiation

We now instantiate quantifications over $\mathbb{Z}$ with a finite set and turn them into finite conjunctions. Instantiating quantifiers is sound and very common in decision procedures and SMT solvers.[5] Although we postpone the full elaboration and proof of completeness to §4,

---

[3] One can also perform a more precise (but more expensive) analysis by checking the validity of Δ', defined as $\neg\psi_2 \implies \bigwedge_{(u,v,w)\in E\setminus F} \delta(u) + w = \delta(v)$, where ψ_2 is the instantiation of ψ_1 constructed in Section 3.5. This classifies more goals as tangle-free. The validity of Δ' depends on the choice of F and reference nodes.

[4] To give more hints, the algorithm may also identify the off-forest edge which causes the unprovable equation, then reconstruct and print the nonzero cycle—see §8.

[5] Replacing arbitrary quantifiers over Z in negative positions with finitely many instantiations is sound (if the resulting formula is valid, then the original formula is also valid), but not complete (the opposite direction as soundness). [4]

the basic idea of this conversion is that it is enough to only consider finite crucial points in the arrays instead of an infinity of indices. The crucial points divide arrays into segments. The indices in the same segment share the same constraints, so they can be represented by a close crucial point.

For each connected component of the IPG, we first calculate an index set.

Definition 2 (Index Set) The index set I is a set of integer terms for a connected component constructed as follows:

1. Initialize I with the empty set;
2. For every array variable a in the component, add $-1-\delta(a)$ and $|a|-\delta(a)$ to I;
3. For every occurrence $a[n]$, where n is an expression without universal quantified variables, add $n-\delta(a)$ to I;
4. For every occurrence $i+n \leq m$ or $m \leq i+n$ in index guards, where i is a universally quantified variable and n, m are terms without quantified variables, add $m-n-\delta(i)$ to I;

For an integer term k, define $I+k$ as $\{j+k \mid j \in I\}$. Now, in ψ_1, we replace every quantification $\forall \vec{\mathbf{i}} : \mathbb{Z}^n .\, F(\vec{\mathbf{i}})$ with $\bigwedge_{\vec{\mathbf{e}} \in I+\delta(\vec{\mathbf{i}})} F(\vec{\mathbf{e}})$, yielding the formula ψ_2, where the notation $\vec{\mathbf{e}} \in I+\delta(\vec{\mathbf{i}})$ means $\{\vec{\mathbf{e}} \mid \forall t.e_t \in I+\delta(i_t)\}$. That is, quantification over $\mathbb{Z}$ has been replaced by conjunction over a finite set. The number of conjuncts is $|I|^{|\vec{\mathbf{i}}|}$.

In the first subgoal of the running example, $\delta(i) = \delta(a) = \delta(j) = 0$ and $\delta(b) = -|a|$. Component 1 contains a and i and Component 2 contains b and j. The index sets are

$$
\begin{aligned}
I_1 &= \{-1, |a|\} \cup \{\} \cup \{0, |a|-1\} \\
I_2 &= \{-1+|a|, |b|+|a|\} \cup \{k+|a|\} \cup \{|a|, |a|+|b|-1\}.
\end{aligned}
$$

I_1 and I_2 are written as unions of three sets. These sets are terms added in steps 2, 3 & 4, respectively. And we instantiate i with I_1 and j with I_2 (because $\delta(i) = \delta(j) = 0$), and we get

$$
\begin{aligned}
&0 \leq k < |b| \implies \\
&\qquad \bigwedge_{i \in I_1} (0 \leq i < |a| \implies P(a[i])) \implies \\
&\qquad \bigwedge_{j \in I_2} (|a| \leq j < |a|+|b| \implies P(b[j-|a|])) \implies 0 \leq k < |b|+|a| \implies P(b[k]).
\end{aligned}
$$

3.6 Decision

The algorithm invokes the base solver $\mathcal{O}_{\mathrm{B}}$ as follows. For each array variable a, create a fresh uninterpreted function f_a from integer to the element sort of a to represent the content of a, and a fresh integer variable L_a to represent the length of a. Replace terms of the form $a[i]$ with $f_a(i)$ and replace terms of the form $|a|$ with L_a. Because the lengths of arrays are nonnegative, add an assumption $L_a \geq 0$ for each variable L_a. Because out-of-bounds access yields d_S, add an assumption $i < 0 \vee i \geq L_a \implies f_a(i) = d_S$ for each distinct occurrence of $f_a(i)$. So

$$\psi_{\mathrm{B}} \quad := \quad (\text{assumptions}) \implies (\psi_2 \text{ in the base theory}).$$

The resulting formula ψ_{B} is quantifier-free in the base theory with uninterpreted functions, which falls in the range of the base solver $\mathcal{O}_{\mathrm{B}}$. So we use $\mathcal{O}_{\mathrm{B}}$ to decide its validity.

For the first subgoal of the running example, the formula sent to the base solver is

$$
\begin{aligned}
& L_a \geq 0 \wedge L_b \geq 0 \implies \bigwedge_{i \in I_1} (i < 0 \vee i \geq L_a \implies f_a(i) = d_S) \implies \\
& \quad \bigwedge_{j \in I_2} (j - L_a < 0 \vee j - L_a \geq L_b \implies f_b(j - L_a) = d_S) \implies \\
& \quad (k < 0 \vee k \geq L_b \implies f_b(k) = d_S) \implies \\
& \quad 0 \leq k < L_b \implies \\
& \quad \bigwedge_{i \in I_1} (0 \leq i < L_a \implies P(f_a(i))) \implies \\
& \quad \bigwedge_{j \in I_2} (L_a \leq j < L_a + L_b \implies P(f_b(j - L_a))) \implies 0 \leq k < L_b + L_a \implies P(f_b(k)).
\end{aligned}
$$

4 Correctness

In this section, we show that the algorithm in §3 is correct, i.e. sound and complete.

Proving soundness of the algorithm is more straightforward. The rewrites in §3.1–3.3 all use provably correct equations; the versions of those rules that we use in our Coq implementation are all proved as Coq lemmas. The classification in §3.4 is sound as it only renames variables. Instantiating quantifiers in negative positions as in §3.5 makes ψ_2 have weaker assumptions than ψ_1, so it is sound. The following lemma shows §3.6 is sound and complete, which will also be used later in the completeness proof.

Lemma 1 *Given a quantifier-free formula ψ_2 in the array theory, let ψ_B be the base theory formula constructed as in §3.6, then ψ_2 and ψ_B are equivalid. Furthermore, given a counterexample M_B of ψ_B, we can construct a counterexample M_2 of ψ_2, and vice versa.*

Proof For each model M_2 in the array theory, there is a corresponding model M_B in T_B, such that $M_\mathsf{B}[\![L_a]\!] = M_2[\![|a|]\!]$ and $M_\mathsf{B}[\![f_a(i)]\!] = M_2[\![a[i]]\!]$ for $0 \leq i < |a|$. This is a one-to-one correspondence between models in the array theory and models in T_B that satisfy $L_a \geq 0$ and where accessing f_a out-of-bounds yields d_S. For each pair of corresponding models M_2 and M_B, model M_2 satisfies ψ_2 if and only if M_B satisfies ψ_2's correspondence in T_B (the conclusion part of ψ_B). A model in T_B is a counterexample of ψ_B if and only if it satisfies the additional assumptions that $L_a \geq 0$ and accessing f_a out-of-bounds yields d_S, and it falsifies ψ_2's correspondence in T_B. So other models in T_B cannot be counterexamples of ψ_B, and the one-one correspondence above is also a one-one correspondence between counterexamples of M_2 and M_B, which completes the proof.

Our Coq implementation uses these principles to generate a formal correctness proof for every solved goal.

Completeness. If a tangle-free formula ψ is valid, then our algorithm will prove it, by reducing it to a valid ψ_{base}. The most difficult part of the proof is to prove the finite instantiation in §3.5 is correct. The main idea for proving this step is that given a concrete index set S, we can delegate each element in an array to the nearest element in the index set. So we define the projection function:

Definition 3 (Projection Function) For a nonempty finite integer set S, the projection function $\text{proj}_S : \mathbb{Z} \to S$ is defined as

$$\text{proj}_S(n) = \begin{cases} \min S, & \text{if } n < \min S, \\ \max\{m \in S \mid m \leq n\}, & \text{otherwise.} \end{cases}$$

An intuitive understanding of the projection function is that we partition integers into a finite number of segments and project each segment to a representative element, so we only need to consider a finite set of representatives, instead of all integers.

The projection function proj_S is indeed projective, in the sense that for any $n \in S$, we have $\text{proj}_S(n) = n$. The projection function is monotone, i.e. for all integers n and m,

$$n \leq m \implies \text{proj}_S(n) \leq \text{proj}_S(m).$$

Another property is the shifting transformation, that for all integers n and m,

$$\text{proj}_{S+m}(n) = \text{proj}_S(n-m) + m,$$

where $S + m = \{x + m \mid x \in S\}$. This is because if $n < \min(S+m)$, then $\min(S+m) = \min S + m$ and $n - m < \min S$, so both sides are $\min S + m$. Otherwise

$$\text{proj}_{S+m}(n) = \max\{x \in S+m \mid x \leq n\} = \max\{x \mid x - m \in S \wedge x \leq n\},$$
$$\text{proj}_S(n-m) + m = \max\{y \in S \mid y \leq n-m\} + m = \max\{y + m \mid y \in S \wedge y \leq n-m\}.$$

They are equal because $\{x \mid x - m \in S \wedge x \leq n\}$ and $\{y + m \mid y \in S \wedge y \leq n - m\}$ are the same set by taking $x := y + m$.

Outline of completeness proof. Suppose ψ is tangle-free (i.e. Δ is valid[6]), and ψ_{B} is created by transforming ψ by the algorithm in §3. We will show that given any counterexample M_{B} for ψ_{B}, we can construct a counterexample M for ψ. First, Lemma 1 shows that we can construct a counterexample M_2 for ψ_2.

Remark. Since M_2 is a counterexample for ψ_2, as Δ is valid, then M_2 satisfies

$$\bigwedge_{(u,v,w) \in E \setminus F} \delta(u) + w = \delta(v).$$

Because δ is constructed by propagating through F, we have M_2 satisfies

$$\bigwedge_{(u,v,w) \in E} \delta(u) + w = \delta(v). \tag{8}$$

Definition 4 (Construction of counterexamples) Let ψ be tangle-free. Let ψ_2 be created by instantiating ψ as in §3.5. Suppose M_2 is a counterexample for ψ_2. Then we will construct M (a counterexample for ψ) as follows.

[6] or Δ' is valid; see footnote 3

For every nonarray variable to which M_2 gives an interpretation, we let M give the same interpretation. (The only difference between ψ and ψ_2 is that ψ_2 interrogates arrays at a subset of the indices at which ψ does, but scalar variables are treated the same.)

For the interpretation of array variables in M: let I be the index set defined in Definition 2, which is a set of terms. For every array variable a, the value in M is defined as

$$a = [a_0, \dots, a_{M_2[\![|a|]\!]-1}] \quad \text{where} \quad a_i = M_2[\![a]\!]\left[\text{proj}_{M_2[\![I+\delta(a)]\!]}(i)\right].$$

Remark. Our implementation does not construct models, since it is focused on deductive proofs. But in an SMT setting where it is useful to present counterexamples, the complexity of constructing M will be only $|I|$ times the number of array variables.

Lemma 2 *For every term e in ψ_1 such that e does not contain quantified variables and the sort of e is not an array, then $M[\![e]\!] = M_2[\![e]\!]$.*

Proof The proof is by strong induction on the structure of e. So we prove that $M[\![e]\!] = M_2[\![e]\!]$ by assuming every nonarray proper subterm of e, denoted by e', has $M[\![e']\!] = M_2[\![e']\!]$. If e is a nonarray variable/constant/literal, M and M_2 have the same values for it, so the equation holds. If e is an application of a nonarray function/predicate, the interpretation of the function/predicate is also the same in M and M_2. Then by the induction hypothesis on subterms $M[\![e]\!] = M_2[\![e]\!]$. If e is in form of $|a|$ for an array variable a, by the definition of M, we have $M[\![|a|]\!] = M_2[\![|a|]\!]$.

If e is in form of $a[n]$ for an array variable a and an integer expression n, by the induction hypothesis, $M[\![n]\!] = M_2[\![n]\!]$. According to the construction of the index set I, term $n - \delta(a)$ is in I. Because the projection function is projective, $\text{proj}_{M_2[\![I+\delta(a)]\!]}(M_2[\![n]\!]) = M_2[\![n]\!]$. For the term $a[n]$, if $M[\![n]\!]$ is out of bounds, $M[\![a[n]]\!] = d_S = M_2[\![a[n]]\!]$. Otherwise,

$$M[\![a[n]]\!] = M_2[\![a]\!]\left[\text{proj}_{M_2[\![I+\delta(a)]\!]}(M_2[\![n]\!])\right] = M_2[\![a]\!]\,[M_2[\![n]\!]] = M_2[\![a[n]]\!].$$

So we conclude $M[\![a[n]]\!] = M_2[\![a[n]]\!]$. The enumeration above contains all kinds of terms, so we have $M[\![e]\!] = M_2[\![e]\!]$ for all terms e by induction.

Corollary 1 *$M[\![I]\!] = M_2[\![I]\!]$, $M[\![\delta(a)]\!] = M_2[\![\delta(a)]\!]$ and $M[\![\delta(i)]\!] = M_2[\![\delta(i)]\!]$, for every array variable a and every quantified variable i.*

Throughout the proof, the projection function will only be applied with S being I with a shift of $\delta(a)$ or $\delta(i)$ in assignment M (or M_2, which is the same). When dealing with quantifiers, we will use an assignment M' that extends M with quantified variables. So we use the shorthands

$$\begin{aligned}
\text{proj}^{M'}(e) &:= \text{proj}_{M[\![I]\!]}(M'[\![e]\!]),\\
\text{proj}_a^{M'}(e) &:= \text{proj}_{M[\![I+\delta(a)]\!]}(M'[\![e]\!]),\\
\text{proj}_i^{M'}(e) &:= \text{proj}_{M[\![I+\delta(i)]\!]}(M'[\![e]\!]).
\end{aligned}$$

Theorem 1 (Completeness) *Let ψ_B be created by processing ψ as in §3. For tangle-free ψ, if ψ_B is invalid, then so is ψ. Furthermore, if we have a counterexample M_B for ψ_B, we can construct a counterexample M for ψ.*

Proof Because (as mentioned in §3.2 and §3.3) the rewrite rules are sound and complete, ψ_1 is equivalent to ψ, so we only need to consider ψ_1. Construct M_2 from M_{B} as in Lemma 1 and M as in Definition 4. Then M_2 is a counterexample of ψ_2. By Lemma 2, the quantifier-free formulas in ψ_1 (which are the same in ψ_2) have the same truth values in M and M_2. So to prove M is a counterexample of ψ_1, it is enough to consider the quantified formulas. Recall that each quantified formula φ in ψ_1,

$$\varphi \quad = \quad \forall \vec{\mathbf{i}} : \mathbb{Z}.\ \varphi_I(\vec{\mathbf{i}}) \implies \varphi_V(\vec{\mathbf{i}}),$$

is converted into φ_2 in ψ_2,

$$\varphi_2 \quad = \quad \bigwedge_{\{\vec{\mathbf{e}} \mid \forall t. e_t \in I + \delta(i_t)\}} \Big(\varphi_I(\vec{\mathbf{e}}) \implies \varphi_V(\vec{\mathbf{e}}) \Big).$$

Because these formulas only appear in negative positions, we only need to show that if M_2 satisfies φ_2, then M satisfies φ.

Let $(i_1, \dots, i_k)$ be the elements of $\vec{\mathbf{i}}$. Let M' be any extension of M by binding $i_1, \dots, i_k$ to arbitrary integer values. We need to show M' satisfies $\varphi_I(\vec{\mathbf{i}}) \implies \varphi_V(\vec{\mathbf{i}})$. For convenience, we define a notation for applying the projection function to a sequence of variables:

$$\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}) = \Big(\mathrm{proj}_{i_1}^{M'}(i_1), \dots, \mathrm{proj}_{i_k}^{M'}(i_k) \Big).$$

Since $\mathrm{proj}_{i_t}^{M'}(i_t) \in M_2(I + \delta(i_t))$, there is a vector $\vec{\mathbf{e}}$ such that $\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}) = M_2(\vec{\mathbf{e}})$ and a conjunct of φ_2 is $(\varphi_I(\vec{\mathbf{e}}) \implies \varphi_V(\vec{\mathbf{e}}))$. So M_2 satisfies the conjunct and then also satisfies

$$\varphi_I(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}})) \implies \varphi_V(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}})).$$

Because $\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}})$ is a constant vector, this is a quantifier-free formula whose nonconstant terms are in ψ_1, so M also satisfies it:

$$M[\![\varphi_I(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}))]\!] \implies M[\![\varphi_V(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}))]\!].$$

To prove $M'[\![\varphi_I(\vec{\mathbf{i}})]\!] \implies M'[\![\varphi_V(\vec{\mathbf{i}})]\!]$, it is enough to show

$$M'[\![\varphi_I(\vec{\mathbf{i}})]\!] \implies M[\![\varphi_I(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}))]\!] \tag{9}$$

and

$$M[\![\varphi_V(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}))]\!] \implies M'[\![\varphi_V(\vec{\mathbf{i}})]\!]. \tag{10}$$

We show (9) *as follows.* Because φ_I is a positive Boolean combination of atoms, it is enough to show that, for every atom φ_{atom} in φ_I,

$$M'[\![\varphi_{\text{atom}}(\vec{\mathbf{i}})]\!] \implies M[\![\varphi_{\text{atom}}(\mathrm{proj}_{\vec{\mathbf{i}}}^{M'}(\vec{\mathbf{i}}))]\!]. \tag{11}$$

$\varphi_{\text{atom}}(\vec{\mathbf{i}})$ has three possible forms. We analyze these three forms as follows, using n and m for terms without universally quantified variables and using i and j for universally quantified variables.

1. If $\varphi_{\text{atom}}(\vec{\mathbf{i}})$ is of the form $n \le m$, then $\varphi_{\text{atom}}(\vec{\mathbf{i}})$ does not depend on $\vec{\mathbf{i}}$ and (11) is straightforward.

2. If $\varphi_{\text{atom}}(\vec{\mathbf{i}})$ is of the form $i+n \le m$ or $i+n \ge m$, we first consider the case $i+n \le m$. We can replace φ_{atom} with $i \le m-n$, and (11) is reduced to

$$M'[\![i]\!] \le M[\![m-n]\!] \implies \text{proj}_i^{M'}(i) \le M[\![m-n]\!]. \tag{12}$$

The construction of index set I guarantees that $M[\![m-n-\delta(i)]\!] \in M[\![I]\!]$. So $M[\![m-n]\!] \in M[\![I]\!] + M[\![\delta(i)]\!]$, and then $M[\![m-n]\!] = \text{proj}_i^{M'}(m-n)$. By monotonicity of the projection function, (12) holds. The proof is the same for $i+n \ge m$.

3. If $\varphi_{\text{atom}}(\vec{\mathbf{i}})$ is in the form of $i+n \le j+m$, then we may replace φ_{atom} with $i-j \le m-n$, and (11) is reduced to

$$M'[\![i-j]\!] \le M[\![m-n]\!] \implies \text{proj}_i^{M'}(i) - \text{proj}_j^{M'}(j) \le M[\![m-n]\!].$$

There is an edge $(i, j, n-m)$ in the IPG—because $\varphi_{\text{atom}}(\vec{\mathbf{i}})$ is part of the formula from which the IPG is constructed, and Definition 1 will have constructed this edge. Then, by (8), $M[\![\delta(i)]\!] + M[\![n-m]\!] = M[\![\delta(j)]\!]$. So $M[\![\delta(i) - \delta(j)]\!] = M[\![m-n]\!]$ and it is enough to show

$$M'[\![i-j]\!] \le M[\![\delta(i) - \delta(j)]\!] \implies \text{proj}_i^{M'}(i) - \text{proj}_j^{M'}(j) \le M[\![\delta(i) - \delta(j)]\!].$$

By moving terms and expansion of denotation, it is enough to show

$$\begin{aligned} &M'[\![i]\!] - M[\![\delta(i)]\!] \le M'[\![j]\!] - M[\![\delta(j)]\!] \\ &\qquad\qquad \implies \text{proj}_i^{M'}(i) - M[\![\delta(i)]\!] \le \text{proj}_j^{M'}(j) - M[\![\delta(j)]\!]. \end{aligned}$$

By definition and then by the properties of the projection function,

$$\text{proj}_i^{M'}(i) = \text{proj}_{I+\delta(i)}^{M'}(i) = \text{proj}_I^{M'}(i - \delta(i)) + M[\![\delta(i)]\!].$$

Then we have

$$\text{proj}_i^{M'}(i) - M[\![\delta(i)]\!] = \text{proj}_I^{M'}(i - \delta(i)).$$

The same result also holds for j, so it is enough to show

$$M'[\![i]\!] - M[\![\delta(i)]\!] \le M'[\![j]\!] - M[\![\delta(j)]\!] \implies \text{proj}_I^{M'}(i - \delta(i)) \le \text{proj}_I^{M'}(j - \delta(j)),$$

which follows from monotonicity of the projection function.

To prove (10)*,* recall (from the definition of the *value constraint* in §2) that these universally quantified variables only occur as indices in the form of $a[i+n]$, where n is a term without quantified variables. The construction of a's value in M implies $M'[\![a[i+n]]\!] = M[\![a[\text{proj}_a^{M'}(i+n)]]\!]$ when $M'[\![i+n]\!]$ is in bounds. When $M'[\![i+n]\!]$ is out of bounds, because $\{-1, |a|\} \subseteq I + \delta(a)$, the projected index $\text{proj}_a^{M'}(i+n)$ is also out of bounds. So in either case, we always have $M'[\![a[i+n]]\!] = M[\![a[\text{proj}_a^{M'}(i+n)]]\!]$. The occurrence of $a[i+n]$ implies an IPG edge (i, a, n). By (8), we have $M[\![\delta(i)+n]\!] = M[\![\delta(a)]\!]$. So by the shifting transformation,

$$\begin{aligned} \text{proj}_i^{M'}(i) + M[\![n]\!] &= \text{proj}_{I+\delta(i)}^{M'}(i) + M[\![n]\!] = \text{proj}_I^{M'}(i - \delta(i)) + M[\![\delta(i)]\!] + M[\![n]\!] \\ &= \text{proj}_I^{M'}(i + n - \delta(a)) + M[\![\delta(a)]\!] = \text{proj}_{I+\delta(a)}^{M'}(i+n) = \text{proj}_a^{M'}(i+n). \end{aligned}$$

So $M'[\![a[i+n]]\!] = M[\![a[\text{proj}_i^{M'}(i) + n]]\!]$. By replacing every subterm of the form $a[i+n]$ in $\varphi_V(\vec{\mathbf{i}})$ with $a[\text{proj}_i^{M'}(i) + n]$, (10) holds. Hence, Theorem 1 holds.

5 Discussion on entanglement

Why rewriting is needed. One could calculate an IPG *before* step 3 of the algorithm—that is, without rewriting the quantified formulas. But this IPG would be less precise, classifying more goals as entangled. The method would directly calculate the positions in arrays accessed by universal variables. For example, in the property formula $\forall i : \mathbb{Z}.\ P((a \cdot b)[i])$, the indexing $(a \cdot b)[i]$ indicates either $a[i]$ or $b[i - |a|]$. So we *could* construct an IPG without rewriting using the following procedure written in OCaml-like pseudocode for indexing of the form $e[i+n]$, where e is an array expression.

let addEdge $i\ n\ e$ =
  match e **with**
  | array variable $a \rightarrow$ connect i and a with edges labelled n and $(-n)$
  | $x^k \rightarrow ()$
  | $e_1 \cdot e_2 \rightarrow$ addEdge $i\ e_1\ n$; addEdge $i\ e_2\ (n + |e_2|)$
  | slice$(e_1, j, k) \rightarrow$ addEdge $i\ e_1\ (i + j)$
  | map$_f(e_1, \ldots, e_k) \rightarrow$ addEdge $i\ n\ e_1$; $\cdots$; addEdge $i\ n\ e_k$

Unfortunately, this IPG construction without rewriting and pruning is more likely to have nonzero cycles. For example, if there is an assumption

$$\forall i.\ 0 \leq i < |a| + |b| \implies P(a \cdot b[i])$$

constructing the IPG without rewriting will produce edges $(i, a, 0)$ and $(i, b, -|a|)$. So a and b are connected. But after rewriting into two separated assumptions and renaming the quantified variable with i and j respectively, the IPG edges $(i, a, 0)$ and $(j, b, -|a|)$ will not connect a and b. So it will be less likely to be entangled when there are other assumptions.

Another way to avoid rewriting. Our rewrite rules simplify the app operator. One could imagine a different method: for each term of the form $a \cdot b$, replace with a fresh variable c and add the assumption $c = a \cdot b$ as

$$|c| = |a| + |b| \ \bigwedge\ \forall i.\ 0 \leq i < |a| \implies c[i] = a[i]$$
$$\bigwedge\ \forall j.\ 0 \leq j < |b| \implies c[j + |a|] = b[j].$$

Then we leave the elimination of c to the base solver $\mathcal{O}_{\mathsf{B}}$, and we would not need rewriting. Unfortunately, this method classifies many more goals as entangled. It implies edges in the IPG: $(i, c, 0)$, $(i, a, 0)$, $(j, c, |a|)$, $(j, b, 0)$. If there is another edge, or path, from a to b and the shift is 0, there is a nonzero cycle $(a, i, 0), (i, c, 0), (c, j, -|a|), (j, b, 0), (b, a, 0)$. The advantage of rewriting is avoiding these additional edges and nonzero cycles.

Comparison with string solvers. Our classification of tangle-free and entangled goals is not arbitrary—it is similar to nonoverlapping [25] and acyclicity [1,2] restrictions in the literature of string solvers—but not identical, as we now explain.

Recall the example discussed in the introduction, $a \cdot b_1 = b_2 \cdot a$. If $|a| > |b_2|$, then a has a periodic structure, $a = a_0 a_0 \ldots a_0 a_1$. Such periodic structure is the difficulty in using this assumption to prove the conclusion.

A more complicated example is, $\quad a \cdot c_1 = b \cdot c_2 \quad c_3 \cdot b = c_4 \cdot a.$
The first equation shows a and b have a common prefix of length $\min(|a|, |b|)$, while the

second shows a and b have a common suffix of the same length. If $|b| < |a|$, then a has a common prefix and suffix, which is the same case as the first example. If $|a| < |b|$, we have the same for b.

In our theory we would express the first example as $|a| + |b_1| = |b_2| + |a|$ and

$$\forall i.\ 0 \le i < |a| + |b_1| \implies (a \cdot b_1)[i] = (b_2 \cdot a)[i].$$

After rewriting the quantified formula, one of the branches is

$$\forall i.\ (0 \le i < |a| \land |b_2| \le i < |b_2| + |a|) \implies a[i] = a[i - |b_2|].$$

So the IPG has edges $(i, a, 0)$ and $(a, i, |b_2|)$, which form a nonzero cycle. The second example is expressed as length equations and

$$\forall i.\ 0 \le i < |a| + |c_1| \implies (a \cdot c_1)[i] = (b \cdot c_2)[i] \tag{13}$$

$$\forall i.\ 0 \le i < |c_3| + |a| \implies (c_3 \cdot a)[i] = (c_4 \cdot b)[i]. \tag{14}$$

After rewriting, one branch of (13) is $\forall i.\ (0 \le i < |a| \land 0 \le i < |b|) \implies a[i] = b[i]$. One of the branches of (14) is

$$\forall j.\ (0 \le j - |c_3| < |a| \land 0 \le j - |c_4| < |b|) \implies a[j - |c_3|] = b[j - |c_4|].$$

There are IPG edges $(i, a, 0)$, $(i, b, 0)$, $(j, a, -|c_3|)$, $(j, b, -|c_4|)$. Unless $|a| = |b|$ (note that we have $|c_3| + |a| = |c_4| + |b|$), there is a nonzero cycle, $(i, a, 0)$, $(a, j, |c_3|)$, $(j, b, -|c_4|)$, $(b, i, 0)$, of accumulated shift $|c_3| - |c_4|$. In these two examples, our classification of tangle-free and entangled goals captures the same difficulty as the nonoverlapping and acyclic conditions.

The major difference between our solver and those string solvers is that we manipulate string equations by extensionality, i.e. converting to a quantified equation for elements. This allows expressing more complicated elementwise relations between sequences, including transducers that can be expressed as elementwise relations. Comparing with regular expressions, elementwise relations can only express a part of regular expressions, but can use predicates from other theories (including uninterpreted functions/predicates for all element sorts but not array sorts) and communicate with them, to express properties such as "the elements in an integer sequence are in a certain range."

Abdulla's acyclic condition [1] is less liberal than our nonzero acyclicity: their graph has no offset-labels, so they cannot distinguish nonzero cycles from zero-offset cycles.

The conditions proposed in [2] and [25] are not covered by our condition, but the power of those solvers comes from case splitting. Both those solvers split a into $a_1 \cdot a_2$ if there is an equation $a \cdot b = c \cdot d$ and $|c| < |a|$, to split the equation into $a_1 = c$ and $a_2 \cdot b = d$. If the relationship between $|a|$ and $|c|$ is unknown, those solvers split into three cases, namely $|a| < |c|$, $|a| = |c|$ and $|a| > |c|$. This makes the solvers more powerful, but less efficient, especially when there are constraints like $a \cdot b \cdot c \cdot d \cdot e = f \cdot g \cdot h \cdot i \cdot j$, there will be $\binom{8}{4}$ cases, even without counting the cases with concatenation points sharing same positions.

The algorithm described by Zheng et al. [25] was embodied in the Z3str2 solver. We tested the two successors of this algorithm, Z3str3 and Z3str4.

We compared our solver with Z3str3 and Z3str4 on this goal, which requires quite a bit of case splitting:

$$0 \le n < |a \cdot b \cdot c \cdot d \cdot e| - 1 \land a \cdot b \cdot c \cdot d \cdot e = f \cdot g \cdot h \cdot i \cdot j \implies$$
$$\mathrm{slice}(n, n+1, a \cdot b \cdot c \cdot d \cdot e) \cdot \mathrm{slice}(n+1, n+2, a \cdot b \cdot c \cdot d \cdot e) = \mathrm{slice}(n, n+2, f \cdot g \cdot h \cdot i \cdot j).$$

Our solver (see §6) proved this goal in 412s.[7] We did not include this example in our benchmark suite, since it was not motivated by a real verification—it is meant to illustrate how the different algorithms handle case-splitting.

Our algorithm does less case-splitting on this goal. Although we would have expected Zheng's algorithm to solve this goal, the implementation Z3str3 reports *unknown* after several seconds. Z3str4's behavior is unpredictable: in 10 trials, it succeeded 4 times in 1.5s on average, reported unknown 5 times in 6.6s on average, did not terminate 1 time. Our solver is more reliable on this kind of goal.

6 Implementation

We implemented a solver using our algorithm in the built-in tactic language, Ltac, of the Coq proof assistant, and installed it into distribution 2.7 of the Verified Software Toolchain's VST-Floyd prover [8].

We choose the combination of linear integer arithmetic (LIA) and uninterpreted functions (UF) as the base theory, while allowing customization with other theories by providing additional theory solvers. The algorithm requires a complete base solver, but Coq does not provide a combined decision procedure for LIA+UF. So we use a heuristic base solver, which compromises the completeness of our implementation—it may not be able to solve some goals that our *algorithm* can solve, because the base solver cannot solve the goal after reduction. In some cases, the user must use our solver to reduce the original goal and manually prove some remaining subgoals (see "with help" in Fig. 6).

Arrays are represented by inductive lists and array operators are implemented as functions. The default value d_S appears as an implicit argument of the nth function, and is filled automatically using a typeclass ($\texttt{Inhabitant}\ S$). The inference rules for array operators are proved correct as lemmas in Coq. For efficiency,[8] we add update as a primitive, using inference rules derived from identities:

$$|\text{update}(i,a,x)| = |a| \quad \text{and} \quad \text{update}(j,a,x)[i] = \text{if } i = j \text{ then } x \text{ else } a[i].$$

The solver takes a proof goal expressed as a series of assumptions and a conclusion. Each assumption can be a logical combination of integer equations and inequalities, a (dis-)equality of terms with uninterpreted functions, an uninterpreted predicate (a literal of the form $P(x,\dots)$ or $\neg P(x,\dots)$), or a quantified array property formula. The solver destructs the existentially quantified assumptions by introducing fresh variables. If the conclusion is of the form $A \wedge B$, the solver proves them separately. If the conclusion has a universal quantifier after conversion, e.g. equation or weakening relation between arrays, an element is not in an array, or an array is sorted, the solver converts the conclusion into its quantified version and puts the quantified variables and their ranges into the assumptions by the `intros` tactic.

Our solver treats goals on three levels: length level, quantifier-free fragment level, and array property level.

The *length solver* solves linear integer equations and inequalities involving array lengths. The length solver is frequently called because all inference rules for array elements require

[7] As discussed in §7, we believe our *algorithm* is fast but our *implementation* is slow because of the use of Ltac.

[8] Two-case $i = j$, $i \neq j$ case splits, rather than three-case $i < j$, $i = j$, $i > j$.

length facts, so it needs to be efficient. The length of the same array term may be used repeatedly, so the length solver maintains a table to cache length results. It calls the linear integer arithmetic solver in Coq to prove equations and inequalities.

The *quantifier-free fragment solver* solves goals that do not require instantiating quantifiers in the assumptions. We implement a rewrite procedure to apply the inference rules for the quantifier-free fragment as shown in Fig. 3 (in addition with the rule for update). We preferentially apply rules that produce fewer subgoals—that is, only one subgoal (such as LENGTH_REPEAT), or where all but one subgoal is immediately provable. An example of the latter is NTH_REPEAT in the case where $0 \le i < n$ is provable, the second subgoal is immediately provable by contradiction. We use the *length solver* in such tests, e.g. for the last subgoal of NTH_APP.

When there are no branches to rewrite, the solver picks a term of the form $(a{\cdot}b)[i]$ or $\text{update}(i,x,a)[i]$ and performs case splitting before continuing rewriting and repeating this procedure until there are no more array operators other than length and nth.

The *array property solver* is called, when the first two levels fail to prove the goal. This solver proves the goal using quantified formulas in assumptions. Our Ltac implementation handles only a limited set of quantified forms with at most two quantified variables and at most two different array positions accessed by quantified variables each. These special forms are,

$$\begin{aligned}
\text{forall_range}(l,r,a,\varphi) &: \forall i.\ l \le i < r \implies \varphi(a[i]) \\
\text{forall_range2}(l,r,d,a,b,\varphi) &: \forall i.\ l \le i < r \implies \varphi(a[i], b[i+d]) \\
\text{forall_triangle}(l_1,r_1,l_2,r_2,d,a,b,\varphi) &: \forall i\ j.\ l_1 \le i < r_1 \wedge l_2 \le j < r_2 \wedge i \le j+d \\
&\qquad\qquad \implies \varphi(a[i], b[j])
\end{aligned}$$

These three forms cover common predicates, e.g. elementwise predicate, equality and sortedness. And any bounded tangle-free domain of one or two indices defined by the index guard clauses can be covered by a set of quantified formulas in these three forms. The solver reduces over array operations until the array terms are variables, by applying the rules in Fig. 4 and putting the additional index constraint with min/max with the original bounds. If the range of an index can be proved to be empty, that quantified assumption will be removed. Then the solver calculates whether the index propagation graph has nonzero cycles, by calculating the index shift δ for each array (index shifts for quantified variables are not maintained—we can find them from the arrays they use). If the goal is found to be entangled, the solver reports it. If not, it instantiates the quantified formulas using the indices that appear in the assumptions and the conclusion. Due to the lack of a combined (LIA + uninterp. funcs) decision procedure, the solver tries to determine whether the index clause holds in the instance, and performs case splitting if it cannot be determined.

The implemented solver also allows custom reduction rules to support more array operations for the quantifier-free fragment (see, *e.g.*, † in Fig. 6). Common operations, e.g. reversal, duplicating each element, creating an array of an integer range, and casting an array of 32-bit integers into an array of bytes, can be represented by length and indexing reductions.

		Our algo	Our solver	Dafny	Z3/Z3str3
C functions	strcat	yes	yes	(yes)	yes
	strcpy	yes	yes	(yes)	yes
	strlen	yes	yes	(yes)	yes
	strcmp	yes**	with help	yes	no
	reverse	yes†	yes†	yes*	N/A
General goals	in_app	yes	yes	(yes)	(yes)
	not_in_app	yes	yes	(yes)	(yes)
	assoc_list	yes**	with help	yes	no
	sorted_rotate	yes	yes	no	no
††	drop_drop	yes	yes	yes	yes
††	car_drop_is_nth	yes	yes	yes	yes
††	disjoint_unappend	yes	yes	no	no
† † †	slice_of_slice	yes	yes	no	yes

"with help", see second para. of §6;

(yes) we did not verify that Dafny/Z3 can solve these goals;

† with extension, see last para. of §6.

*Dafny can solve the goal expressed with quantifiers and relations, but not when expressed with functions.
**Assessed by running our solver to step 6, then checking semimanually that the remaining goals are in LIA+UIF.
†† These goals are proved (manually) as lemmas in the VigNAT proof [24]; see footnote 2.
† † † This goal, a generalization of drop_drop, is from Dafny's proposed standard library, https://github.com/dafny-lang/libraries/blob/master/src/Collections/Sequences/Seq.dfy, July 2021.

Fig. 6: Verification power

7 Evaluation

We evaluate the *utility* and the *efficiency* of the solver by experiments.

Verification power. We test the power of our array solver using C functions and general proof goals. The results are shown in Fig. 6. The C functions are provided with human-written loop invariants; VST generates verification conditions by symbolic execution. We test whether the array solver can solve the verification conditions. We compare with similar functions written in Dafny, with corresponding invariants. We manually translated the verification conditions of C string functions and tested them in Z3str3. Z3str3 cannot solve some verification conditions in strcmp that involve quantifiers to express string equations and null-terminated strings (without internal nulls).

We also test the solver with some other proof goals. The most difficult proof goals are assoc_list and sorted_rotate. The assoc_list example arises from reasoning about association lists: given key sort K and value sort V, an association list is a list (or array) of key-value pairs, i.e. $A(K \times V)$. Searching an association list is finding the first key-value pair whose key is a given key k. We can define the search using an auxiliary function,

$$\mathsf{get_index}(l,k) := \begin{cases} 0, & l = [(k,v_0),\dots] \text{ or } l = [] \\ 1 + \mathsf{get_index}(l',k), & l = [(k_0,v_0)]\cdot l' \text{ and } k \neq k_0 \end{cases}$$

A useful lemma to reason about association lists is

$$\forall lki.\ \mathsf{get_index}(l,k) = i \iff \begin{array}{l}(0 \leq i < |l| \wedge \mathsf{fst}(l[i]) = k \wedge k \notin \mathrm{map}_{\mathsf{fst}}(\mathrm{slice}(0,i,l))) \\ \vee (i = |l| \wedge k \notin \mathrm{map}_{\mathsf{fst}}(\mathrm{slice}(0,i,l))).\end{array}$$

where $x \notin l$ is expressed as $\forall i.\ 0 \leq i < |l| \implies l[i] \neq x$.

```
predicate sorted2(s: seq<int>)
{ forall j, k :: 0 <= j <= k < |s| ==> s[j] <= s[k] }

lemma sorted_rotate(s1: seq<int>, s2: seq<int>, s3: seq<int>, N: int)
  requires forall i :: 0 <= i < |s1| ==> 0 <= s1[i] < N
  requires forall i :: 0 <= i < |s2| ==> 0 <= s2[i] < N
  requires sorted2(s1 + s2)
  requires |s1| == |s3|
  requires forall i :: 0 <= i < |s1| ==> s3[i] == s1[i] + N
  ensures sorted2(s2 + s3)
{
  assert forall i :: 0 <= i < |s1| ==> (s1 + s2)[i] == s1[i];
  assert forall i :: 0 <= i < |s2| ==> (s1 + s2)[i + |s1|] == s2[i];
}
```

Fig. 7: Proof for sorted_rotate in Dafny. If Dafny could solve this fully automatically, the body (in braces) would be empty.

To prove this lemma, we use an induction on l. In the case that $l = [(k_0, v_0)] \cdot l'$ and $k \neq k_0$, we need to prove the induction step

$$\begin{aligned} k \neq k_0 \implies & \quad (\varphi(i,l') \iff \varphi(i+1, [(k_0,v_0)] \cdot l')), \\ \text{where } \varphi(i,l) \quad := & \quad (0 \le i < |l| \wedge \mathsf{fst}(l[i]) = k \wedge k \notin \text{map}_{\mathsf{fst}}(\text{slice}(0,i,l))) \\ & \quad \vee (i = |l| \wedge k \notin \text{map}_{\mathsf{fst}}(\text{slice}(0,i,l))). \end{aligned}$$

This falls within our array theory. With a complete base solver, our algorithm would prove this goal; in our Coq implementation (with no combined theory of LIA + uninterp. functions), our algorithm successfully reduces the goal to simpler goals that are straightforward for the human user.

The other example, sorted_rotate, is this: Suppose we have an nondecreasing sequence $s = s_1 \cdot s_2$ with every element of s between 0 and N. Then the sequence $s_2 \cdot \text{map}_{(+N)}(s_1)$ is also nondecreasing. The proof goal is expressed as,

$$\mathsf{sorted}(s_1 \cdot s_2) \wedge (\forall i. 0 \le i < |s_1 \cdot s_2| \implies 0 \le (s_1 \cdot s_2)[i] < N) \implies \mathsf{sorted}(s_2 \cdot \text{map}_{(+N)}(s_1)).$$

Our solver proves sorted_rotate automatically.

We encode these goals in Dafny's and Z3's sequence theories. Neither supports the map operator, so we encode formulas with map using an auxiliary variable $a_f = \text{map}_f(a)$ and add assumptions

$$|a_f| = |a| \wedge (\forall i.\ 0 \le i < |a| \implies a_f[i] = f(a[i])).$$

Dafny can prove sorted_rotate with two human-written auxiliary assertions as hints, as shown in Fig. 7. The hints are short, but it was not obvious how to find them. In fact, we had to write a 21-line proof and remove unnecessary hints in order to get the two-line proof. Z3 can prove neither of these two hard examples.

When our solver cannot solve a goal because the base solver in Coq is not complete, it needs the user's help, too, as shown in Fig. 6. For example, it cannot solve

$$(\text{assumptions}) \implies \texttt{b2val}(i = |l_1| \wedge i = |l_2|) = \texttt{b2val}(\bot),$$

where `b2val` converts a Boolean to a C value. The user needs to reduce it to a Boolean equation by congruence and then solve it by the list solver. This is simpler than Dafny because the user only needs to handle goals in the base theory.

	manual	solver
strlen	3	2
strcpy	31	2
strcat	45	4
strcmp	74	21
reverse	37	3

(Lines of code of proof script)

Fig. 8: Human effort verifying C functions with and without the array solver

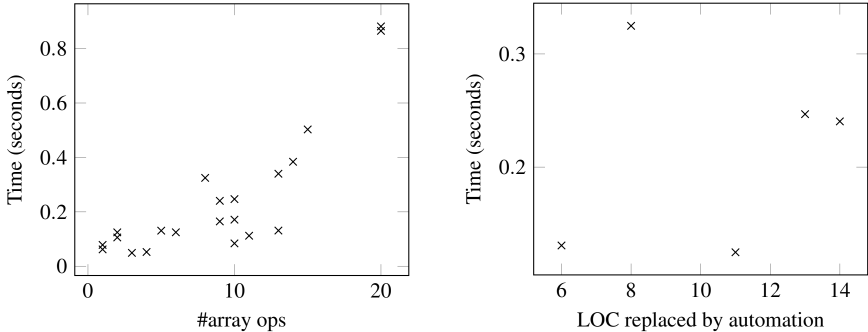


Fig. 9: Performance of quantifier-free goals. We plot time against the difficulty of the goals, as measured by two parameters: (1) the number of array operators in the proof goal, only counting update, app, slice, and map; (2) the lines of code of the handwritten proof-script that users had to write using our previous ad-hoc proof automation for each goal (we only have this data for a subset of goals).

Application in verifying C programs. We evaluate the utility of our solver in verifying functional correctness of C functions, by comparing the correctness proof of a set of samples using VST with and without our array theory solver. The sample set includes four functions from the C standard string library—`string.h`, `strlen`, `strcpy`, `strcat`, and `strcmp`—in addition to `reverse` (which reverses a C array as in Figure 1), to demonstrate the application of custom operators. A proof in VST consists of two parts: symbolic execution proof and model-level proof. The symbolic execution proof is almost completely automatic—the user only needs to use tactics according to the program statements with proper loop invariants and postconditions. The array solver helps the most in the model-level proof. It automates the proof goals to show one array abstraction is stronger than another array abstraction, especially when both array abstractions involve array operators. Fig. 8 shows that our new array solver allows the user to write far fewer lines of Coq proof-script, compared to our previous automation.

Performance. We measured our solver on an Intel i7 10th-generation CPU at 2.30GHz with 16GB memory. Fig. 9 shows the results. Most quantifier-free goals are solved within 1 second, and goals with array properties are solved in about 2 seconds. Based on other empirical performance studies of solvers implemented in Coq [5]—comparing the same algorithm in C++ versus Coq—Fig. 9 suggests that our *algorithm* may be an order of magnitude better than Dafny's. That is, our solver is about 2x slower than Dafny, but our implementation is in Ltac rather than C++ and ours generates Coq proof witnesses for every goal.

Goal	#∀	LOC	Our solver(s)	Dafny(s)
strcmp_loop_if1	2		0.51	
strcmp_loop_exit1	1		0.15	
strcmp_loop	3	29	1.29	1.06
sorted_rotate_array_prop	3		2.10	1.26
row_clear_sound1	1		0.12	
frame_insert_sound1	2		0.19	
frame_query_sound2	2		0.21	
filter_refresh'_sound_frame_sim1	2		2.42	

Fig. 10: Performance on quantified goals. "#∀" indicates the number of quantified array accesses. For example, $\forall i.\ P(a[i], b[i])$ and $\forall i j.\ P(a[i], a[j])$ both count as 2. We suggest "#∀" is the main factor for quantified goals' difficulty.

8 Diagnosis

When a decision procedure or semidecision procedure does not find a proof, then it is polite for it to demonstrate a counterexample or otherwise hint at why no proof was found. Our implementation in Coq does not give counterexamples, but in some cases it does give useful explanations.

When a length-equality hypothesis is missing and unprovable: In the last proof goal shown in Figure 1, there are two premises, the second of which is $|b| = |c|$. If that hypothesis was missing from the proof goal, then the goal is not valid. But it may have been that $|b| = |c|$ would have been provable, but the user has abstracted b or c before proving it; or it may be that c is a complex expression whose length is not analyzable fully automatically.

In such a case, our solver detects that it *must* have $|b| = |c|$, and it prints the following error message:

> [The theory solver] cannot solve this goal. Try asserting above the line, a hypothesis that will help prove $|b| = |c|$.

At which point, it's often straightforward for the user to assert and prove $|b| = |c|$ before retrying the array theory solver.

When entanglement is detected: For a goal such as Example 2, the solver responds,

> List solver cannot solve this goal because it is entangled. That is, some assumption(s) relate some slice of an array to an overlapping slice of the *same* array, which sets up a recurrence. The list_solve tactic does not attempt to solve such recurrences, which in general are undecidable.
> Entangled: because 0 + 1 = 0 is not provable.

Or, in Example 3, "because $l + n - m = 0$ is not provable." If the user can prove that fact and retry the solver, it will succeed.

Other goals: In other cases where no proof is found, the message is,

> list_solve cannot solve this goal; list_simplify can sometimes diagnose where subgoals need extra assistance.

The list_simplify tactic runs our algorithm to step 6, then leaves for the user any goals where "invoke the base solver" fails. This leaves the Coq proof state in a form readable by the user, who can examine the failed subgoals and (sometimes) determine which extra facts should be proved before trying the solver again.

9 Array solver as an SMT theory solver

So far, we have described the theory of arrays with concatenation as a single theory and have given a solver for the tangle-free fragment. However, modern SMT solvers utilize the Nelson-Oppen procedure to combine theories. It is infeasible to directly use the array solver as a theory solver in Nelson-Oppen, because Nelson-Oppen adds additional array equality constraints and makes the goal entangled. In this section, we show how our solver can be used as a theory solver in a general SMT solver and can combine with other theories. This is useful when only a part of the query is related to arrays and array elements, or there are uninterpreted functions whose arguments or results are arrays. For example, for uninterpreted functions $f : \mathbb{Z} \to A(S)$ and $P : A(S) \to \mathbb{P}$, we want to prove that

$$P\Big(f(1)\cdot\big(f(2)\cdot f(3)\big)\Big) \implies P\Big(\big(f(1)\cdot f(2)\big)\cdot f(3)\Big). \tag{15}$$

The work in this section is a bit speculative, as we have only proved it correct, but not implemented it. We extend the classification of tangle-free and entangled into the combined theory. Then we run the normal Nelson-Oppen algorithm. But when Nelson-Oppen invokes the array solver, the latter manipulates the equations from Nelson-Oppen by removing the equations that are not in a set predetermined by the array solver with respect to the original formula. Then the array solver determines the satisfiablity of the manipulated constraints and reports to Nelson-Oppen. Although the array solver is cheating Nelson-Oppen, we show that the latter can still determine satisfiability correctly. If a model is needed, we provide a modified Nelson-Oppen to generate a model.

In this section, we describe the problem as satisfiability instead of validity. Let T_0 be the theory of uninterpreted functions, T_1 be the theory of arrays with concatenation for some base theory, and $T_2, \dots, T_n$ be other theories of interest. Without loss of generality, we take the union of the sorts of all the theories as the sort set. Different theories T_i and T_j must not share any functions other than equality for each sort. The theory T_i may not have any functions involving any array sorts including equations, for $i \geq 2$.

Recall that each theory is a class of interpretations. The combined theory $T = \bigoplus_{i=0}^{n} T_i$ is the class of interpretations whose Σ_i-reduct is isomorphic to an interpretation in T_i for every i. The standard Nelson-Oppen procedure is correct with respect to this definition of combined theory [4]. Quantified array formulas, including array equations, are handled inside the array solver, so we abstract them as special predicates when talking about the Nelson-Oppen procedure, so they are "quantifier-free". The Nelson-Oppen procedure first purifies the query into a series of φ_i, each of which is a "quantifier-free" formula in the theory T_i. So φ_i does not have array equations (or inequalities), for $i \geq 2$. Assume φ_0 is in negation normal form (NNF). The literals of φ_0 that are array equations and inequalities must take the form $a \bowtie b$, $a \bowtie f(\dots)$ or $f(\dots) \bowtie g(\dots)$, where $\bowtie \in \{=, \neq\}$, a and b are array variables, and $f(\dots)$ and $g(\dots)$ are uninterpreted functions applied on any terms. We can replace $f(\dots) \bowtie g(\dots)$ with $a \bowtie g(\dots)$ for fresh a and add $a = f(\dots)$ to conjoin with φ_0, so we can eliminate this case.

If the SMT solver is a DPLL(T) algorithm [4], it is even simpler: we need only consider conjunctions of literals. In that case, we only need to consider the case $a = f(\dots)$, because other cases can be reduced to this case by adding auxiliary variables and moving literals without uninterpreted functions to φ_1. But here we consider the general case. (We will use a, b to range over array variables, and x, y to range over general terms.)

The outline of the algorithm is as follows. After splitting the formula, we construct the IPG with edges E_1 for φ_1 (array theory) as usual. Then we analyze φ_0 (equality with un-

interpreted functions) and add some additional edges E_0. After that, we test IPG $E_1 \cup E_0$ entanglement, and raise an error if it is entangled. Then, we let Nelson-Oppen nondeterministically choose an equivalence relation R for the shared variables. Since the theory is many-sorted, it needs to pick an equivalence relation R_S for each sort. The choice procedure is untouched—the Nelson-Oppen implementation can be used unmodified, with any choice heuristics it uses.

As usual in Nelson-Oppen, define the arrangement

$$ar(R) = \bigwedge_{S \text{ is a sort}} \left(\bigwedge_{(x,y) \in R_S} x = y \wedge \bigwedge_{(x,y) \notin R_S} x \neq y \right).$$

We say a formula φ is compatible with $ar(R)$ if $\varphi \wedge ar(R)$ is satisfiable. The Nelson-Oppen procedure then queries the array solver whether φ_1 is compatible with $ar(R)$. The array solver redefines the constraint as

$$ar'(R) = \bigwedge_{S \text{ is a sort}} \left(\bigwedge_{(x,y) \in R_S \,\wedge\, (S \notin \{\text{array sorts}\} \vee (x,y,0) \in E_0)} x = y \wedge \bigwedge_{(x,y) \notin R_S} x \neq y \right). \tag{16}$$

The array solver decides whether φ_1 is compatible with $ar'(R)$.[9] This is tangle-free, because we have constructed the IPG and decided it is tangle-free, and for every array equation $x = y$ in $ar'(R)$, there is an edge $(x,y,0)$ in $E_1 \cup E_0$. The model of $\varphi_1 \wedge ar'(R)$ implies an equivalence relation R'. For $i \geq 2$, any model of φ_i satisfies either both R and R' or neither of them, because φ_i does not have array sorts. We prove that if φ_0 is compatible with $ar(R)$ then φ_0 is compatible with $ar(R')$ (see Lemma 3). So the original formula is satisfiable if and only if there exists R, such that φ_1 is compatible with $ar'(R)$ and φ_i is compatible with $ar(R)$, for $i \neq 1$.

We construct the edges E_0 as follows. In order to satisfy φ_0, consider uninterpreted functions, which can be characterized by congruence: $x \neq y \vee f(x) = f(y)$. (It is straightforward to generalize to multiargument functions.) So we only need to think about these two kinds of constraints. A constraint of the form $x \neq y$ does not cause any IPG edges, because even if x and y are array variables, $x \neq y$ is considered as quantifier-free in the array solver.[10] For constraints of the form $f(x) = f(y)$, we do need to add IPG edges.

We first examine in φ_0 the literals of the form $a = f(\dots)$, where a is an array variable. If there are $a = f(\dots)$ and $b = f(\dots)$, to satisfy φ_0 we potentially need $a = b$, so we add an edge $(a,b,0)$ to E_0. For the literals of the form $a = b$ in φ_0, we also add an edge $(a,b,0)$. Adding edges is not needed for literals $a \neq b$ and $a \neq f(\dots)$. For the example in (15), we need to add (a spanning tree of) edges between the variables that denote $f(1)$, $f(2)$ and $f(3)$, but we do not add edges between the variables that denote the arguments of P. Entanglement is determined on the IPG with the edges $E_1 \cup E_0$.

When the Nelson-Oppen procedure queries the array solver with an equivalence relation R, the array solver ignores the equations that do not correspond to edges in E_0 as in (16). If the array solver determines that the query is satisfiable, that means the model that it uses to satisfy φ_1 may have a different equivalence relation R' for shared array variables. The following lemma shows that φ_0 is compatible with $ar(R')$ if it is compatible with $ar(R)$.

---

[9] We can decide it because the correctness results in Section 4 can be generalized to the case that the IPG has more edges than the queried formula $\varphi_i \wedge ar'(R)$.

[10] In the array solver, $a \neq b$ is treated as a shorthand for $|a| \neq |b| \vee \exists i.\ 0 \leq i < |a| \wedge a[i] \neq b[i]$. The existentially quantified i can be treated as a free variable, so it is quantifier-free.

Lemma 3 *For any φ_0 in T_0 in NNF, whose array literals are in the form of $a \bowtie b$ and $a \bowtie f(\dots)$ for $\bowtie \in \{=,\neq\}$, for any equivalence relations R and R' over shared variables, such that R' is a subrelation of R, and R' and R only differ for array-variable pairs (a,b) without edges in E_0 between them: if $\varphi_0 \wedge ar(R)$ is satisfiable then $\varphi_0 \wedge ar(R')$ is also satisfiable.*

Proof It is enough to consider that φ_0 is a conjunction of literals, since there is a clause in the disjunctive normal form (DNF) of φ_0 that is compatible with $ar(R)$, and we only need to prove the same clause is compatible with $ar(R')$. We then eliminate uninterpreted functions. Pick an uninterpreted function f (whose arguments and result sorts do not need to be arrays), and for all its occurrences $f(x)$ in φ_0, where x is any term, introduce a fresh variable f_x for $f(x)$ and for each pair of f_x and f_y, add $x \neq y \ \vee \ f_x = f_y$ to the conjunction of φ_0. Repeat this procedure until all uninterpreted functions are eliminated. Let φ_0' be the DNF of the result. We have that, for any equivalence relation R, $\varphi_0' \wedge ar(R)$ is equisatisfiable with $\varphi_0 \wedge ar(R)$. For the same reason that we consider φ_0 as a conjunction of literals, it is enough to consider each clause c_0' of φ_0', which is a conjunction of equation and inequality literals. So we can consider each sort separately. The nonarray sort literals are same between R and R', so we only need to consider array literals. These literals are either from a literal of φ_0 (after eliminating uninterpreted functions) or $x \neq y$ or $f_x = f_y$ added as above. In c_0', if there is an array equation $f_x = t$, we can substitute f_x with t and eliminate f_x. Repeat this procedure until all the literals of the form $f_x = t$ are eliminated. If, for f_x, there are array inequalities $f_x \neq t_1, \dots, f_x \neq t_k$, because array sorts have infinitely many elements, it is always possible to find a value for f_x not equal to any of the t_i, so we can remove these inequalities. So now all variables in c_0' are original variables in φ_0. For each equation literal $a = b$ in c_0', it is either an original equation in φ_0, or it is added because there are $a = f_x$ and $b = f_y$, which are $a = f(\dots)$ and $b = f(\dots)$ in φ_0. In either case, there is an IPG edge between a and b. Therefore if $(a,b) \in R$, then $(a,b) \in R'$, too. The inequality literals in c_0' are satisfied by R' because R' is a subrelation of R. So c_0' is compatible with R implies c_0' is compatible with R'. That concludes the proof.

When the Nelson-Oppen procedure finds an equivalence relation R that is claimed compatible with each φ_i by each theory solver, it will report SAT as usual. Although the array solver might actually loosen the constraint, there exists R' that is compatible with each φ_i. If it needs to generate a model, the solver of uninterpreted functions must use the equivalence relation R' given by the array solver to generate its model.

In summary, in order to support theory combination, we need to add additional edges to the IPG only for uninterpreted functions with array results. And we use the Nelson-Oppen procedure off-the-shelf with a loosened but still sound interpretation of the result.

10 Conclusion

We have defined a theory of arrays with concatenation, and we have given a procedure that classifies goals into tangle-free and entangled and decides validity of tangle-free goals. We have implemented the procedure in Coq/Ltac, and found it is very useful for semi-automated program verifiers: it significantly reduces human effort in functional-correctness verification, and it can give clear feedback.

In the future, we hope to (1) implement our algorithm in an SMT solver, (2) investigate other applications of the theory, and (3) generalize the theory to multidimensional arrays and arrays of arrays.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *International Conference on Computer Aided Verification*, pages 150–166. Springer, 2014. https://doi.org/10.1007/978-3-319-08867-9_10.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. Chain-free string constraints. In *International Symposium on Automated Technology for Verification and Analysis*, pages 277–293. Springer, 2019. https://doi.org/10.1007/978-3-030-31784-3_16.
3. Andrew W Appel. Verified software toolchain. In *European Symposium on Programming*, pages 1–17. Springer, 2011. https://doi.org/10.1007/978-3-642-19718-5_1.
4. Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. https://doi.org/10.1007/978-3-319-10575-8_11.
5. Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular smt proofs for fast reflexive checking inside coq. In *International Conference on Certified Programs and Proofs*, pages 151–166. Springer, 2011. https://doi.org/10.1007/978-3-642-25379-9_13.
6. Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. An smt-lib format for sequences and regular expressions. *SMT*, 12:76–86, 2012.
7. Aaron R Bradley, Zohar Manna, and Henny B Sipma. What's decidable about arrays? In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006. https://doi.org/10.1007/11609773_28.
8. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, June 2018. https://doi.org/10.1007/s10817-018-9457-5.
9. Taolue Chen, Yan Chen, Matthew Hague, Anthony W Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017. https://doi.org/10.1145/3158091.
10. Przemysław Daca, Thomas A Henzinger, and Andrey Kupriyanov. Array folds logic. In *International Conference on Computer Aided Verification*, pages 230–248. Springer, 2016. https://doi.org/10.1007/978-3-319-41540-6_13.
11. Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: what's decidable? In *Haifa Verification Conference*, pages 209–226. Springer, 2012. https://doi.org/10.1007/978-3-642-39611-3_21.
12. Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*, pages 306–320. Springer, 2009. https://doi.org/10.1007/978-3-642-02658-4_25.
13. Peter Habermehl, Radu Iosif, and Tomáš Vojnar. What else is decidable about integer arrays? In *International Conference on Foundations of Software Science and Computational Structures*, pages 474–489. Springer, 2008. https://doi.org/10.1007/978-3-540-78499-9_33.
14. Lukáš Holík, Petr Janků, Anthony W Lin, Philipp Rümmer, and Tomáš Vojnar. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–32, 2017. https://doi.org/10.1145/3158092.
15. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011. https://doi.org/10.1007/978-3-642-20398-5_4.
16. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010. https://doi.org/10.1007/978-3-642-17511-4_20.
17. Anthony W Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–136, 2016. https://doi.org/10.1145/2837614.2837641.
18. Gennadiy Semenovich Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 145(2):147–236, 1977.
19. John McCarthy. Towards a mathematical science of computation. In *Program Verification*, pages 35–56. Springer, 1993. https://doi.org/10.1007/978-94-011-1793-7_2.
20. Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967.
21. Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, May 2004. https://doi.org/10.1145/990308.990312.
22. Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 467–476. ACM, 2006. https://doi.org/10.1145/1132516.1132584.

23. Aaron Stump, Clark W Barrett, David L Dill, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE, 2001.
24. Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*, pages 141–154, 2017.
25. Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017. https://doi.org/10.1007/s10703-016-0263-6.