Reasoning About Vectors using an SMT Theory of Sequences[⋆]

Ying Sheng[1], Andres Nötzli[1], Andrew Reynolds[2], Yoni Zohar[3], David Dill[4], Wolfgang Grieskamp[4], Junkil Park[4], Shaz Qadeer[4], Clark Barrett[1], and Cesare Tinelli[2]

[1]Stanford University [2]The University of Iowa [3]Bar-Ilan University [4]Meta Novi

Abstract. Dynamic arrays, also referred to as vectors, are fundamental data structures used in many programs. Modeling their semantics efficiently is crucial when reasoning about such programs. The theory of arrays is widely supported but is not ideal, because the number of elements is fixed (determined by its index sort) and cannot be adjusted, which is a problem, given that the length of vectors often plays an important role when reasoning about vector programs. In this paper, we propose reasoning about vectors using a theory of sequences. We introduce the theory, propose a basic calculus adapted from one for the theory of strings, and extend it to efficiently handle common vector operations. We prove that our calculus is sound and show how to construct a model when it terminates with a saturated configuration. Finally, we describe an implementation of the calculus in cvc5 and demonstrate its efficacy by evaluating it on verification conditions for smart contracts and benchmarks derived from existing array benchmarks.

1 Introduction

Generic vectors are used in many programming languages. For example, in C++'s standard library, they are provided by `std::vector`. Automated verification of software systems that manipulate vectors requires an efficient and automated way of reasoning about them. Desirable characteristics of any approach for reasoning about vectors include: (i) expressiveness—operations that are commonly performed on vectors should be supported; (ii) generality—vectors are always "vectors of" some type (e.g., vectors of integers), and so it is desirable that vector reasoning be integrated within a more general framework; solvers for satisfiability modulo theories (SMT) provide such a framework and are widely used in verification tools (see [5] for a recent survey); (iii) efficiency—fast and efficient reasoning is essential for usability, especially as verification tools are increasingly used by non-experts and in continuous integration.

Despite the ubiquity of vectors in software on the one hand and the effectiveness of SMT solvers for software verification on the other hand, there is not currently a clean way to represent vectors using operators from the SMT-LIB standard [3]. While the theory of arrays can be used, it is not a great fit because arrays have a fixed size determined by their index type. Representing a dynamic array thus requires additional modeling work. Moreover, to reach an acceptable level of expressivity, quantifiers are needed, which often makes the reasoning engine less efficient and robust. Indeed, part of the

[⋆] This work was funded in part by the Stanford Center for Blockchain Research, NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF), and Meta Novi. Part of the work was done when the first author was an intern at Meta Novi.

motivation for this work was frustration with array-based modeling in the Move Prover, a verification framework for smart contracts [24] (see Section 6 for more information about the Move Prover and its use of vectors). The current paper bridges this gap by studying and implementing a native theory of *sequences* in the SMT framework, which satisfies the desirable properties for vector reasoning listed above.

We present two SMT-based calculi for determining satisfiability in the theory of sequences. Since the decidability of even weaker theories is unknown (see, e.g., [9, 15]), we do not aim for a decision procedure. Rather, we prove model and solution soundness (that is, when our procedure terminates, the answer is correct). Our first calculus leverages techniques for the theory of strings. We generalize these techniques, lifting rules specific to string characters to more general rules for arbitrary element types. By itself, this base calculus is already quite effective. However, it misses opportunities to perform high-level vector-based reasoning. For example, both reading from and updating a vector are very common operations in programming, and reasoning efficiently about the corresponding sequence operators is thus crucial. Our second calculus addresses this gap by integrating reasoning methods from array solvers (which handle reads and updates efficiently) into the first procedure. Notice, however, that this integration is not trivial, as it must handle novel combinations of operators (such as the combination of update and read operators with concatenation) as well as out-of-bounds cases that do not occur with ordinary arrays. We have implemented both variants of our calculus in the cvc5 SMT solver [2] and evaluated them on benchmarks originating from the Move prover, as well as benchmarks that were translated from SMT-LIB array benchmarks.

As is typical, both of our calculi are agnostic to the sort of the elements in the sequence. Reasoning about sequences of elements from a particular theory can then be done via theory combination methods such as Nelson-Oppen [18] or polite combination [16, 20]. The former can be done for stably infinite theories (and the theory of sequences that we present here is stably infinite), while the latter requires investigating the politeness of the theory, which we expect to do in future work.

The rest of the paper is organized as follows. Section 2 includes basic notions from first-order logic. Section 3 introduces the theory of sequences and shows how it can be used to model vectors. Section 4 presents calculi for this theory and discusses their correctness. Section 5 describes the implementation of these calculi in cvc5. Section 6 presents an evaluation comparing several variations of the sequence solver in cvc5 and Z3. We conclude in Section 7 with directions for further research.

Related work: Our work crucially builds on a proposal by Bjørner et al. [8], but extends it in several key ways. First, their implementation (for a logic they call `QF_BVRE`) restricts the generality of the theory by allowing only bit-vector elements (representing characters) and assuming that sequences are bounded. In contrast, our calculus maintains full generality, allowing unbounded sequences and elements of arbitrary types. Second, while our core calculus focuses only on a subset of the operators in [8], our implementation supports the remaining operators by reducing them to the core operators, and also adds native support for the update operator, which is not included in [8].

The base calculus that we present for sequences builds on similar work for the theory of strings [6, 17]. We extend our base calculus to support array-like reasoning based on the weak-equivalence approach [10]. Though there exists some prior work on extending the theory of arrays with more operators and reasoning about length [1, 12, 14], this work does not include support for most of the of the sequence operators we consider here.

The SMT-solver Z3 [11] also provides a solver for sequences. However, its documentation is limited [7], it does not support update directly, and its internal algorithms are not described in the literature. Furthermore, as we show in Section 6, the performance of the Z3 implementation is generally inferior to our implementation in cvc5.

2 Preliminaries

We assume the usual notions and terminology of many-sorted first-order logic with equality (see, e.g., [13] for a complete presentation). We consider many-sorted signatures Σ, each containing a set of sort symbols (including a Boolean sort Bool), a family of logical symbols $\approx$ for equality, with sort $\sigma \times \sigma \to \mathsf{Bool}$ for all sorts σ in Σ and interpreted as the identity relation, and a set of interpreted (and sorted) function symbols. We assume the usual definitions of well-sorted terms, literals, and formulas as terms of sort Bool. A literal is *flat* if it has the form $\bot$, $p(x_1, \ldots, x_n)$, $\neg p(x_1, \ldots, x_n)$, $x \approx y$, $\neg x \approx y$, or $x \approx f(x_1, \ldots, x_n)$, where p and f are function symbols and x, y, and $x_1, \ldots, x_n$ are variables. A Σ-interpretation $\mathcal{M}$ is defined as usual, satisfying $\mathcal{M}(\bot) = \mathrm{false}$ and assigns: a set $\mathcal{M}(\sigma)$ to every sort σ of Σ, a function $\mathcal{M}(f) : \mathcal{M}(\sigma_1) \times \ldots \times \mathcal{M}(\sigma_n) \to \mathcal{M}(\sigma)$ to any function symbol f of Σ with arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$, and an element $\mathcal{M}(x) \in \mathcal{M}(\sigma)$ to any variable x of sort σ. The satisfaction relation between interpretations and formulas is defined as usual and is denoted by $\models$.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$, in which Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, closed under variable reassignment. The *models* of T are the interpretations in $\mathbf{I}$ without any variable assignments. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. Given a (set of) terms S, we write $\mathcal{T}(S)$ to denote the set of all subterms of S. For a theory $T = (\Sigma, \mathbf{I})$, a set S of Σ-formulas and a Σ-formula φ, we write $S \models_T \varphi$ if every interpretation $\mathcal{M} \in \mathbf{I}$ that satisfies S also satisfies φ. By convention and unless otherwise stated, we use letters w, x, y, z to denote variables and s, t, u, v to denote terms.

The theory $T_{\mathsf{LIA}} = (\Sigma_{\mathsf{LIA}}, \mathbf{I}_{T_{\mathsf{LIA}}})$ of *linear integer arithmetic* is based on the signature Σ_{LIA} that includes a single sort Int, all natural numbers as constant symbols, the unary $-$ symbol, the binary $+$ symbol and the binary $\leqslant$ relation. When $k \in \mathbb{N}$, we use the notation $k \cdot x$, inductively defined by $0 \cdot x = 0$ and $(m + 1) \cdot x = x + m \cdot x$. In turn, $\mathbf{I}_{T_{\mathsf{LIA}}}$ consists of all structures $\mathcal{M}$ for Σ_{LIA} in which the domain $\mathcal{M}(\mathsf{Int})$ of Int is the set of integer numbers, for every constant symbol $n \in \mathbb{N}$, $\mathcal{M}(n) = n$, and $+$, $-$, and $\leqslant$ are interpreted as usual. We use standard notation for integer intervals (e.g., $[a, b]$ for the set of integers i, where $a \leqslant i \leqslant b$ and $[a, b)$ for the set where $a \leqslant i < b$).

3 A Theory of Sequences

We define the theory T_{Seq} of sequences. Its signature Σ_{Seq} is given in Figure 1. It includes the sorts Seq, Elem, Int, and Bool, intuitively denoting sequences, elements, integers, and Booleans, respectively. The first four lines include symbols of Σ_{LIA}. We write $t_1 \bowtie t_2$,

Symbol	**Arity**	**SMT-LIB**	**Description**
n	Int	n	All constants $\mathtt{n} \in \mathbb{N}$
$+$	Int × Int → Int	+	Integer addition
$-$	Int → Int	-	Unary Integer minus
$\leqslant$	Int × Int → Bool	<=	Integer inequality
ϵ	Seq	`seq.empty`	The empty sequence
unit	Elem → Seq	`seq.unit`	Sequence constructor
$\lvert\_\rvert$	Seq → Int	`seq.len`	Sequence length
nth	Seq × Int → Elem	`seq.nth`	Element access
update	Seq × Int × Elem → Seq	`seq.update`	Element update
extract	Seq × Int × Int → Seq	`seq.extract`	Extraction (subsequence)
$\_ +\!\!+ \cdots +\!\!+ \_$	Seq × ⋯ × Seq → Seq	`seq.concat`	Concatenation

Fig. 1: Signature for the theory of sequences.

with $\bowtie \in \{>, <, \leqslant\}$, as syntactic sugar for the equivalent literal expressed using $\leqslant$ (and possibly $\neg$). The sequence symbols are given on the remaining lines. Their arities are also given in Figure 1. Notice that $\_ +\!\!+ \cdots +\!\!+ \_$ is a variadic function symbol.

Interpretations $\mathcal{M}$ of T_{Seq} interpret: Int as the set of integers; Elem as some set; Seq as the set of finite sequences whose elements are from Elem; ϵ as the empty sequence; unit as a function that takes an element from $\mathcal{M}(\mathsf{Elem})$ and returns the sequence that contains only that element; nth as a function that takes an element s from $\mathcal{M}(\mathsf{Seq})$ and an integer i and returns the ith element of s, in case i is non-negative and is smaller than the length of s (we take the first element of a sequence to have index 0). Otherwise, the function has no restrictions; update as a function that takes an element s from $\mathcal{M}(\mathsf{Seq})$, an integer i, and an element a from $\mathcal{M}(\mathsf{Elem})$ and returns the sequence obtained from s by replacing its ith element by a, in case i is non-negative and smaller than the length of s. Otherwise, the returned value is s itself; extract as a function that takes a sequence s and integers i and j, and returns the maximal sub-sequence of s that starts at index i and has length at most j, in case both i and j are non-negative and i is smaller than the length of s. Otherwise, the returned value is the empty sequence;[1] $\lvert\_\rvert$ as a function that takes a sequence and returns its length; and $\_ +\!\!+ \cdots +\!\!+ \_$ as a function that takes some number of sequences (at least 2) and returns their concatenation.

Notice that the interpretations of Elem and nth are not completely fixed by the theory: Elem can be set arbitrarily, and nth is only defined by the theory for some values of its second argument. For the rest, it can be set arbitrarily.

3.1 Vectors as Sequences

We show the applicability of T_{Seq} by using it for a simple verification task. Consider the C++ function `swap` at the top of Figure 2. This function swaps two elements in a

[1] In [8], the second argument j denotes the end index, while here it denotes the length of the sub-sequence, in order to be consistent with the theory of strings in the SMT-LIB standard.

vector. The comments above the function include a partial specification for it: if both indexes are in-bounds and the indexed elements are equal, then the function should not change the vector (this is expressed by `s_out==s`). We now consider how to encode the verification condition induced by the code and the specification. The function variables a, b, i, and j can be encoded as variables of sort Int with the same names. We include two copies of s: s for its value at the beginning, and s_{out} for its value at the end. But what should be the sorts of s and s_{out}? In Figure 2 we consider two options: one is based on arrays and the other on sequences.

Example 1 (Arrays). The theory of arrays includes three sorts: index, element (in this case, both are Int), and an array sort Arr, as well as two operators: $x[i]$, interpreted as the ith element of x; and $x[i \leftarrow a]$, interpreted as the array obtained from x by setting the element at index i to a. We declare s and s_{out} as variables of an uninterpreted sort V and declare two functions ℓ and $\mathbf{c}$, which, given v of sort V, return its length (of sort Int) and content (of sort Arr), respectively.[2]

Next, we introduce functions to model vector operations: $\approx_{\mathbf{A}}$ for comparing vectors, $\mathsf{nth}_{\mathbf{A}}$ for reading from them, and $\mathsf{update}_{\mathbf{A}}$ for updating them. These functions need to be axiomatized. We include two axioms (bottom of Figure 2): Ax_1 states that two vectors are equal iff they have the same length and the same contents. Ax_2 axiomatizes the update operator; the result has the same length, and if the updated index is in bounds, then the corresponding element is updated. These axioms are not meant to be complete, but are rather just strong enough for the example.

The first two lines of the `swap` function are encoded as equalities using $\mathsf{nth}_{\mathbf{A}}$, and the last two lines are combined into one nested constraint that involves $\mathsf{update}_{\mathbf{A}}$. The precondition of the specification is naturally modeled using $\mathsf{nth}_{\mathbf{A}}$, and the postcondition is negated, so that the unsatisfiability of the formula entails the correctness of the function w.r.t. the specification. Indeed, the conjunction of all formulas in this encoding is unsatisfiable in the combined theories of arrays, integers, and uninterpreted functions.

The above encoding has two main shortcomings: It introduces auxiliary symbols, and it uses quantifiers, thus reducing clarity and efficiency. In the next example, we see how using the theory of sequences allows for a much more natural and succinct encoding.

Example 2 (Sequences). In the sequences encoding, s and s_{out} have sort Seq. No auxiliary sorts or functions are needed, as the theory symbols can be used directly. Further, these symbols do not need to be axiomatized as their semantics is fixed by the theory. The resulting formula, much shorter than in Example 2 and with no quantifiers, is unsatisfiable in T_{Seq}.

4 Calculi

After introducing some definitions and assumptions, we describe a basic calculus for the theory of sequences, which adapts techniques from previous procedures for the theory

[2] It is possible to obtain a similar encoding using the theory of datatypes; however, here we use uninterpreted functions which are simpler and better supported by SMT solvers.

```
// @pre: 0 <= i,j < s.size() and s[i] == s[j]
// @post: s_out == s
void swap(std::vector<int>& s, int i, int j) {
  int a = s[i];
  int b = s[j];
  s[i] = b;
  s[j] = a;
}
```

	Sequences	**Arrays**
Problem Variables	$a, b, i, j : \mathsf{Int} \quad s, s_{out} : \mathsf{Seq}$	$a, b, i, j : \mathsf{Int} \quad s, s_{out} : V$
Auxiliary Variables		$\ell : V \to \mathsf{Int} \quad \mathbf{c} : V \to \mathsf{Arr}$ $\approx_{\mathbf{A}}: V \times V \to \mathsf{Bool}$ $\mathsf{nth}_{\mathbf{A}} : V \times \mathsf{Int} \to \mathsf{Int}$ $\mathsf{update}_{\mathbf{A}} : V \times \mathsf{Int} \times \mathsf{Int} \to V$
Axioms		$Ax_1 \wedge Ax_2$
Program	$a \approx \mathsf{nth}(s, i) \wedge b \approx \mathsf{nth}(s, j)$ $s_{out} \approx \mathsf{update}(\mathsf{update}(s, i, b), j, a)$	$a \approx \mathsf{nth}_{\mathbf{A}}(s, i) \wedge b \approx \mathsf{nth}_{\mathbf{A}}(s, j)$ $s_{out} \approx_{\mathbf{A}} \mathsf{update}_{\mathbf{A}}(\mathsf{update}_{\mathbf{A}}(s, i, b), j, a)$
Spec.	$0 \leqslant i, j < \lvert s \rvert \wedge \mathsf{nth}(s, i) \approx \mathsf{nth}(s, j)$ $\neg s_{out} \approx s$	$0 \leqslant i, j < \ell(s) \wedge \mathsf{nth}_{\mathbf{A}}(s, i) \approx \mathsf{nth}_{\mathbf{A}}(s, j)$ $\neg s_{out} \approx_{\mathbf{A}} s$

$$Ax_1 := \forall x, y. x \approx_{\mathbf{A}} y \leftrightarrow (\ell(x) \approx \ell(y) \wedge \forall\, 0 \leqslant i < \ell(x). \mathbf{c}(x)[i] \approx \mathbf{c}(y)[i])$$
$$Ax_2 := \forall x, y, i, a. y \approx_{\mathbf{A}} \mathsf{update}_{\mathbf{A}}(x, i, a) \to (\ell(x) \approx \ell(y) \wedge (0 \leqslant i < \ell(x) \to \mathbf{c}(y) \approx \mathbf{c}(x)[i \leftarrow a]))$$

Fig. 2: An example using T_{Seq}.

of strings. In particular, the basic calculus reduces the operators nth and update by introducing concatenation terms. We then show how to extend the basic calculus by introducing additional rules inspired by solvers for the theory of arrays; the modified calculus can often reason about nth and update terms directly, avoiding the introduction of concatenation terms (which are typically expensive to reason about).

Given a vector of sequence terms $\boldsymbol{t} = (t_1, \ldots, t_n)$, we use $\overline{\boldsymbol{t}}$ to denote the term corresponding to the concatenation of $t_1, \ldots, t_n$. If $n = 0$, $\overline{\boldsymbol{t}}$ denotes ϵ, and if $n = 1$, $\overline{\boldsymbol{t}}$ denotes t_1; otherwise (when $n > 1$), $\overline{\boldsymbol{t}}$ denotes a concatenation term having n children. In our calculi, we distinguish between sequence and arithmetic constraints.

Definition 1. *A Σ_{Seq}-formula φ is a* sequence constraint *if it has the form $s \approx t$ or $s \not\approx t$; it is an* arithmetic constraint *if it has the form $s \approx t$, $s \geqslant t$, $s \not\approx t$, or $s < t$ where s, t are terms of sort* Int*, or if it is a disjunction $c_1 \vee c_2$ of two arithmetic constraints.*

Notice that sequence constraints do not have to contain sequence terms (e.g., $x \approx y$ where x, y are Elem-variables). Also, equalities and disequalities between terms of sort Int are both sequence and arithmetic constraints. In this paper we focus on sequence constraints and arithmetic constraints. This is justified by the following lemma. (Proofs of this lemma and later results can be found in an extended version of this paper [23].)

Lemma 1. *For every quantifier-free Σ_{Seq}-formula φ, there are sets $\mathsf{S}_1, \ldots, \mathsf{S}_n$ of sequence constraints and sets $\mathsf{A}_1, \ldots, \mathsf{A}_n$ of arithmetic constraints such that φ is T_{Seq}-satisfiable iff $\mathsf{S}_i \cup \mathsf{A}_i$ is T_{Seq}-satisfiable for some $i \in [1, n]$.*

$$
\begin{array}{ll}
|\epsilon| \to 0 & |\mathsf{unit}(t)| \to 1 \\
|\mathsf{update}(s, i, t)| \to |s| & |s_1 \mathbin{++} \cdots \mathbin{++} s_n| \to |s_1| + \cdots + |s_n| \\
\\
\overline{\boldsymbol{u}} \mathbin{++} \epsilon \mathbin{++} \overline{\boldsymbol{v}} \to \overline{\boldsymbol{u}} \mathbin{++} \overline{\boldsymbol{v}} & \overline{\boldsymbol{u}} \mathbin{++} (s_1 \mathbin{++} \cdots \mathbin{++} s_n) \mathbin{++} \overline{\boldsymbol{v}} \to \overline{\boldsymbol{u}} \mathbin{++} s_1 \mathbin{++} \cdots \mathbin{++} s_n \mathbin{++} \overline{\boldsymbol{v}}
\end{array}
$$

Fig. 3: Rewrite rules for the reduced form $t\!\downarrow$ of a term t, obtained from t by applying these rules to completion.

Throughout the presentation of the calculi, we will make a few simplifying assumptions.

Assumption 1. *Whenever we refer to a set* S *of sequence constraints, we assume:*

1. *for every non-variable term* $t \in \mathcal{T}(\mathsf{S})$*, there exists a variable* x *such that* $x \approx t \in \mathsf{S}$*;*
2. *for every* Seq*-variable* x*, there exists a variable* ℓ_x *such that* $\ell_x \approx |x| \in \mathsf{S}$*;*
3. *all literals in* S *are flat.*

Whenever we refer to a set of arithmetic constraints, we assume all its literals are flat.

These assumptions are without loss of generality as any set can easily be transformed into an equisatisfiable set satisfying the assumptions by the addition of fresh variables and equalities. Note that some rules below introduce non-flat literals. In such cases, we assume that similar transformations are done immediately after applying the rule to maintain the invariant that all literals in $\mathsf{S} \cup \mathsf{A}$ are flat. Rules may also introduce fresh variables k of sort Seq. We further assume that in such cases, a corresponding constraint $\ell_k \approx |k|$ is added to S with a fresh variable ℓ_k.

Definition 2. *Let* C *be a set of constraints. We write* $\mathsf{C} \models \varphi$ *to denote that* C *entails formula* φ *in the empty theory, and write* $\equiv_{\mathsf{C}}$ *to denote the binary relation over* $\mathcal{T}(\mathsf{C})$ *such that* $s \equiv_{\mathsf{C}} t$ *iff* $\mathsf{C} \models s \approx t$*.*

Lemma 2. *For all set* S *of sequence constraints,* $\equiv_{\mathsf{S}}$ *is an equivalence relation; furthermore, every equivalence class of* $\equiv_{\mathsf{S}}$ *contains at least one variable.*

We denote the equivalence class of a term s according to $\equiv_{\mathsf{S}}$ by $[s]_{\equiv_{\mathsf{S}}}$ and drop the $\equiv_{\mathsf{S}}$ subscript when it is clear from the context.

In the presentation of the calculus, it will often be useful to normalize terms to what will be called a *reduced form*.

Definition 3. *Let* t *be a* Σ_{Seq}*-term. The* reduced form *of* t*, denoted by* $t\!\downarrow$*, is the term obtained by applying the rewrite rules listed in Figure 3 to completion.*

Observe that $t\!\downarrow$ is well defined because the given rewrite rules form a terminating rewrite system. This can be seen by noting that each rule reduces the number of applications of sequence operators in the left-hand side term or keeps that number the same but reduces the size of the term. It is not difficult to show that $\models_{T_{\mathsf{Seq}}} t \approx t\!\downarrow$.

We now introduce some basic definitions related to concatenation terms.

Definition 4. *A* concatenation term *is a term of the form* $s_1 \mathbin{++} \cdots \mathbin{++} s_n$ *with* $n \geqslant 2$*. If each* s_i *is a variable, it is a* variable concatenation term*. For a set* S *of sequence constraints, a variable concatenation term* $x_1 \mathbin{++} \cdots \mathbin{++} x_n$ *is* singular *in* S *if* $\mathsf{S} \not\models x_i \approx \epsilon$

for at most one variable x_i with $i \in [1, n]$. A sequence variable x is atomic in S *if* $\mathsf{S} \not\models x \approx \epsilon$ *and for all variable concatenation terms $s \in \mathcal{T}(\mathsf{S})$ such that $\mathsf{S} \models x \approx s$, s is singular in* S.

We lift the concept of atomic variables to atomic representatives of equivalence classes.

Definition 5. *Let S be a set of sequence constraints. Assume a choice function $\alpha : \mathcal{T}(\mathsf{S})/{\equiv_{\mathsf{S}}} \to \mathcal{T}(\mathsf{S})$ that chooses a variable from each equivalence class of $\equiv_{\mathsf{S}}$. A sequence variable x is an* atomic representative in S *if it is atomic in S and $x = \alpha([x]_{\equiv_{\mathsf{S}}})$.*

Finally, we introduce a relation that is the foundation for reasoning about concatenations.

Definition 6. *Let S be a set of sequence constraints. We inductively define a relation $\mathsf{S} \models_{+\!\!+} x \approx s$, where x is a sequence variable in S and s is a sequence term whose variables are in $\mathcal{T}(\mathsf{S})$, as follows:*

1. *$\mathsf{S} \models_{+\!\!+} x \approx x$ for all sequence variables $x \in \mathcal{T}(\mathsf{S})$.*
2. *$\mathsf{S} \models_{+\!\!+} x \approx t$ for all sequence variables $x \in \mathcal{T}(\mathsf{S})$ and variable concatenation terms t, where $x \approx t \in \mathsf{S}$.*
3. *If $\mathsf{S} \models_{+\!\!+} x \approx (\overline{\boldsymbol{w}} +\!\!+ y +\!\!+ \overline{\boldsymbol{z}})\!\downarrow$ and $\mathsf{S} \models y \approx t$ and t is ϵ or a variable concatenation term in S that is not singular in S, then $\mathsf{S} \models_{+\!\!+} x \approx (\overline{\boldsymbol{w}} +\!\!+ t +\!\!+ \overline{\boldsymbol{z}})\!\downarrow$.*

Let α be a choice function for S as defined in Definition 5. We additionally define the entailment relation $\mathsf{S} \models^{*}_{+\!\!+} x \approx \overline{\boldsymbol{y}}$, where $\overline{\boldsymbol{y}}$ is of length $n \geqslant 0$, to hold if each element of $\overline{\boldsymbol{y}}$ is an atomic representative in S and there exists $\overline{\boldsymbol{z}}$ of length n such that $\mathsf{S} \models_{+\!\!+} x \approx \overline{\boldsymbol{z}}$ and $\mathsf{S} \models y_i \approx z_i$ for $i \in [1, n]$.

In other words, $\mathsf{S} \models^{*}_{+\!\!+} x \approx t$ holds when t is a concatenation of atomic representatives and is entailed to be equal to x by S. In practice, t is determined by recursively expanding concatenations using equalities in S until a fixpoint is reached.

Example 3. Suppose $\mathsf{S} = \{x \approx y +\!\!+ z, y \approx w +\!\!+ u, u \approx v\}$ (we omit the additional constraints required by Assumption 1, part 2 for brevity). It is easy to see that u, v, w, and z are atomic in S, but x and y are not. Furthermore, w and z (and one of u or v) must also be atomic representatives. Clearly, $\mathsf{S} \models_{+\!\!+} x \approx x$ and $\mathsf{S} \models x \approx y +\!\!+ z$. Moreover, $y +\!\!+ z$ is a variable concatenation term that is not singular in S. Hence, we have $\mathsf{S} \models_{+\!\!+} x \approx (y +\!\!+ z)\!\downarrow$, and so $\mathsf{S} \models_{+\!\!+} x \approx y +\!\!+ z$ (by using either Item 2 or Item 3 of Definition 6, as in fact $x \approx y +\!\!+ z \in \mathsf{S}$.). Now, since $\mathsf{S} \models_{+\!\!+} x \approx y +\!\!+ z$, $\mathsf{S} \models y \approx w +\!\!+ u$, and $w +\!\!+ u$ is a variable concatenation term not singular in S, we get that $\mathsf{S} \models_{+\!\!+} x \approx ((w +\!\!+ u) +\!\!+ z)\!\downarrow$, and so $\mathsf{S} \models_{+\!\!+} x \approx w +\!\!+ u +\!\!+ z$. Now, assume that $v = \alpha([v]_{\equiv_{\mathsf{S}}}) = \alpha(\{v, u\})$. Then, $\mathsf{S} \models^{*}_{+\!\!+} x \approx w +\!\!+ v +\!\!+ z$.

Our calculi can be understood as modeling abstractly a cooperation between an *arithmetic subsolver* and a *sequence subsolver*. Many of the derivation rules lift those in the string calculus of Liang et al. [17] to sequences of elements of an arbitrary type. We describe them similarly as rules that modify *configurations*.

Definition 7. *A* configuration *is either the distinguished configuration* unsat *or a pair* (S, A) *of a set S of sequence constraints and a set A of arithmetic constraints.*

The rules are given in *guarded assignment form*, where the rule premises describe the conditions on the current configuration under which the rule can be applied, and the conclusion is either unsat, or otherwise describes the resulting modifications to the configuration. A rule may have multiple conclusions separated by $\|$. In the rules, some of the premises have the form $\mathsf{S} \models s \approx t$ (see Definition 2). Such entailments can be checked with standard algorithms for congruence closure. Similarly, premises of the form $\mathsf{S} \models_{\mathsf{LIA}} s \approx t$ can be checked by solvers for linear integer arithmetic.

An application of a rule is *redundant* if it has a conclusion where each component in the derived configuration is a subset of the corresponding component in the premise configuration. We assume that for rules that introduce fresh variables, the introduced variables are identical whenever the premises triggering the rule are the same (i.e., we cannot generate an infinite sequence of rule applications by continuously using the same premises to introduce fresh variables).[3] A configuration other than unsat is *saturated* with respect to a set R of derivation rules if every possible application of a rule in R to it is redundant. A *derivation tree* is a tree where each node is a configuration whose children, if any, are obtained by a non-redundant application of a rule of the calculus. A derivation tree is *closed* if all of its leaves are unsat. As we show later, a closed derivation tree with root node (S, A) is a proof that $\mathsf{A} \cup \mathsf{S}$ is unsatisfiable in T_{Seq}. In contrast, a derivation tree with root node (S, A) and a saturated leaf with respect to all the rules of the calculus is a witness that $\mathsf{A} \cup \mathsf{S}$ is satisfiable in T_{Seq}.

4.1 Basic Calculus

Definition 8. *The calculus* BASE *consists of the derivation rules in Figures 4 and 5.*

Some of the rules are adapted from previous work on string solvers [17, 22]. Compared to that work, our presentation of the rules is noticeably simpler, due to our use of the relation $\models^*_{++}$ from Definition 6. In particular, our configurations consist only of pairs of sets of formulas, without any auxiliary data-structures.

Note that judgments of the form $\mathsf{S} \models^*_{++} x \approx t$ are used in premises of the calculus. It is possible to compute whether such a premise holds thanks to the following lemma.

Lemma 3. *Let* S *be a set of sequence constraints and* A *a set of arithmetic constraints. If* (S, A) *is saturated w.r.t. S-Prop, L-Intro and L-Valid, the problem of determining whether* $\mathsf{S} \models^*_{++} x \approx s$ *for given* x *and* s *is decidable.*

Lemma 3 assumes saturation with respect to certain rules. Accordingly, our proof strategy, described in Section 5, will ensure such saturation before attempting to apply rules relying on $\models^*_{++}$. The relation $\models^*_{++}$ induces a normal form for each equivalence class of $\equiv_{\mathsf{S}}$.

Lemma 4. *Let* S *be a set of sequence constraints and* A *a set of arithmetic constraints. Suppose* (S, A) *is saturated w.r.t. A-Conf, S-Prop, L-Intro, L-Valid, and C-Split. Then, for every equivalence class* e *of* $\equiv_{\mathsf{S}}$ *whose terms are of sort* Seq, *there exists a unique (possibly empty)* $\overline{\boldsymbol{s}}$ *such that whenever* $\mathsf{S} \models^*_{++} x \approx \overline{\boldsymbol{s}'}$ *for* $x \in e$, *then* $\overline{\boldsymbol{s}'} = \overline{\boldsymbol{s}}$. *In this case, we call* $\overline{\boldsymbol{s}}$ *the* normal form *of* e *(and of* x*).*

[3] In practice, this is implemented by associating each introduced variable with a *witness term* as described in [21].

$$\textsf{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} \bot}{\mathsf{unsat}} \qquad \textsf{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{LIA}} s \approx t \qquad s, t \in \mathcal{T}(\mathsf{S})}{\mathsf{S} := \mathsf{S}, s \approx t}$$

$$\textsf{S-Conf}\ \frac{\mathsf{S} \models \bot}{\mathsf{unsat}} \qquad \textsf{S-Prop}\ \frac{\mathsf{S} \models s \approx t \qquad s, t \in \mathcal{T}(\mathsf{S}) \qquad s, t \text{ are } \Sigma_{\mathsf{LIA}}\text{-terms}}{\mathsf{A} := \mathsf{A}, s \approx t}$$

$$\textsf{S-A}\ \frac{x, y \in \mathcal{T}(\mathsf{S}) \cap \mathcal{T}(\mathsf{A}) \qquad x, y : \mathsf{Int}}{\mathsf{A} := \mathsf{A}, x \approx y \quad \| \quad \mathsf{A} := \mathsf{A}, x \not\approx y}$$

$$\textsf{L-Intro}\ \frac{s \in \mathcal{T}(\mathsf{S}) \qquad s : \mathsf{Seq}}{\mathsf{S} := \mathsf{S}, |s| \approx (|s|){\downarrow}} \qquad \textsf{L-Valid}\ \frac{x \in \mathcal{T}(\mathsf{S}) \qquad x : \mathsf{Seq}}{\mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \quad \mathsf{A} := \mathsf{A}, \ell_x > 0}$$

$$\textsf{U-Eq}\ \frac{\mathsf{S} \models \mathsf{unit}(x) \approx \mathsf{unit}(y)}{\mathsf{S} := \mathsf{S}, x \approx y} \qquad \textsf{C-Eq}\ \frac{\mathsf{S} \models^{*}_{+\!+} x \approx \overline{\boldsymbol{z}} \qquad \mathsf{S} \models^{*}_{+\!+} y \approx \overline{\boldsymbol{z}}}{\mathsf{S} := \mathsf{S}, x \approx y}$$

$$\textsf{C-Split}\ \frac{\mathsf{S} \models^{*}_{+\!+} x \approx (\overline{\boldsymbol{w}} +\!\!+\, y +\!\!+\, \overline{\boldsymbol{z}}){\downarrow} \qquad \mathsf{S} \models^{*}_{+\!+} x \approx (\overline{\boldsymbol{w}} +\!\!+\, y' +\!\!+\, \overline{\boldsymbol{z}'}){\downarrow}}{\begin{array}{c} \mathsf{A} := \mathsf{A}, \ell_y > \ell_{y'} \qquad \mathsf{S} := \mathsf{S}, y \approx y' +\!\!+\, k \quad \| \\ \mathsf{A} := \mathsf{A}, \ell_y < \ell_{y'} \qquad \mathsf{S} := \mathsf{S}, y' \approx y +\!\!+\, k \quad \| \\ \mathsf{A} := \mathsf{A}, \ell_y \approx \ell_{y'} \qquad \mathsf{S} := \mathsf{S}, y \approx y' \end{array}}$$

$$\textsf{Deq-Ext}\ \frac{x \not\approx y \in \mathsf{S} \qquad x, y : \mathsf{Seq}}{\begin{array}{c} \mathsf{A} := \mathsf{A}, \ell_x \not\approx \ell_y \quad \| \\ \mathsf{A} := \mathsf{A}, \ell_x \approx \ell_y, 0 \leqslant i < \ell_x \quad \mathsf{S} := \mathsf{S}, w_1 \approx \mathsf{nth}(x, i), w_2 \approx \mathsf{nth}(y, i), w_1 \not\approx w_2 \end{array}}$$

Fig. 4: Core derivation rules. The rules use k and i to denote fresh variables of sequence and integer sort, respectively, and w_1 and w_2 for fresh element variables.

We now turn to the description of the rules in Figure 4, which form the core of the calculus. For greater clarity, some of the conclusions of the rules include terms before they are flattened. First, either subsolver can report that the current set of constraints is unsatisfiable by using the rules A-Conf or S-Conf. For the former, the entailment $\models_{\mathsf{LIA}}$ (which abbreviates $\models_{T_{\mathsf{LIA}}}$) can be checked by a standard procedure for linear integer arithmetic, and the latter corresponds to a situation where congruence closure detects a conflict between an equality and a disequality. The rules A-Prop, S-Prop, and S-A correspond to a form of Nelson-Oppen-style theory combination between the two sub-solvers. The first two communicate equalities between the sub-solvers, while the third guesses arrangements for shared variables of sort Int. L-Intro ensures that the length term $|s|$ for each sequence term s is equal to its reduced form $(|s|){\downarrow}$. L-Valid restricts sequence lengths to be non-negative, splitting on whether each sequence is empty or has a length greater than 0. The unit operator is injective, which is captured by U-Eq. C-Eq concludes that two sequence terms are equal if they have the same normal form. If two sequence variables have different normal forms, then C-Split takes the first differing components y and y' from the two normal forms and splits on their length relationship. Note that C-Split is the source for non-termination of the calculus (see, e.g., [17, 22]). Finally, Deq-Ext handles disequalities between sequences x and y by either asserting that their lengths are different or by choosing an index i at which they differ.

Figure 5 includes a set of reduction rules for handling operators that are not directly handled by the core rules. These reduction rules capture the semantics of these operators by reduction to concatenation. R-Extract splits into two cases: Either the extraction uses

$$\textsf{R-Extract}\ \frac{x \approx \mathsf{extract}(y, i, j) \in \mathsf{S}}{\begin{array}{c}\mathsf{A} := \mathsf{A}, i < 0 \lor i \geqslant \ell_y \lor j \leqslant 0 \qquad \mathsf{S} := \mathsf{S}, x \approx \epsilon \quad \| \\ \mathsf{A} := \mathsf{A}, 0 \leqslant i < \ell_y, j > 0, \ell_k \approx i, \ell_x \approx \min(j, \ell_y - i) \\ \mathsf{S} := \mathsf{S}, y \approx k \mathbin{+\!\!+} x \mathbin{+\!\!+} k'\end{array}}$$

$$\textsf{R-Nth}\ \frac{x \approx \mathsf{nth}(y, i) \in \mathsf{S}}{\begin{array}{c}\mathsf{A} := \mathsf{A}, i < 0 \lor i \geqslant \ell_y \quad \| \\ \mathsf{A} := \mathsf{A}, 0 \leqslant i < \ell_y, \ell_k \approx i \qquad \mathsf{S} := \mathsf{S}, y \approx k \mathbin{+\!\!+} \mathsf{unit}(x) \mathbin{+\!\!+} k'\end{array}}$$

$$\textsf{R-Update}\ \frac{x \approx \mathsf{update}(y, i, z) \in \mathsf{S}}{\begin{array}{c}\mathsf{A} := \mathsf{A}, i < 0 \lor i \geqslant \ell_y \qquad \mathsf{S} := \mathsf{S}, x \approx y \quad \| \\ \mathsf{A} := \mathsf{A}, 0 \leqslant i < \ell_y, \ell_k \approx i, \ell_{k'} \approx 1 \\ \mathsf{S} := \mathsf{S}, y \approx k \mathbin{+\!\!+} k' \mathbin{+\!\!+} k'', x \approx k \mathbin{+\!\!+} \mathsf{unit}(z) \mathbin{+\!\!+} k''\end{array}}$$

Fig. 5: Reduction rules for extract, nth, and update. The rules use k, k', and k'' to denote fresh sequence variables. We write $s \approx \min(t, u)$ as an abbreviation for $s \approx t \lor s \approx u, s \leqslant t, s \leqslant u$.

an out-of-bounds index or a non-positive length, in which case the result is the empty sequence, or the original sequence can be described as a concatenation that includes the extracted sub-sequence. R-Nth creates an equation between y and a concatenation term with $\mathsf{unit}(x)$ as one of its components, as long as i is not out of bounds. R-Update considers two cases. If i is out of bounds, then the update term is equal to y. Otherwise, y is equal to a concatenation, with the middle component (k') representing the part of y that is updated. In the update term, k' is replaced by $\mathsf{unit}(z)$.

Example 4. Consider a configuration (S, A), where S contains the formulas $x \approx y \mathbin{+\!\!+} z$, $z \approx v \mathbin{+\!\!+} x \mathbin{+\!\!+} w$, and $v \approx \mathsf{unit}(u)$, and A is empty. Hence, $\mathsf{S} \models |x| \approx |y \mathbin{+\!\!+} z|$. By L-Intro, we have $\mathsf{S} \models |y \mathbin{+\!\!+} z| \approx |y| + |z|$. Together with Assumption 1, we have $\mathsf{S} \models \ell_x \approx \ell_y + \ell_z$, and then with S-Prop, we have $\ell_x \approx \ell_y + \ell_z \in \mathsf{A}$. Similarly, we can derive $\ell_z \approx \ell_v + \ell_x + \ell_w, \ell_v \approx 1 \in \mathsf{S}$, and so $(*)$ $\mathsf{A} \models_{\mathsf{LIA}} \ell_z \approx 1 + \ell_y + \ell_z + \ell_w$. Notice that for any variable k of sort Seq, we can apply L-Valid, L-Intro, and S-Prop to add to A either $\ell_k > 0$ or $\ell_k = 0$. Applying this to y, z, w, we have that $\mathsf{A} \models_{\mathsf{LIA}} \bot$ in each branch thanks to $(*)$, and so A-Conf applies and we get unsat.

4.2 Extended Calculus

Definition 9. *The calculus* EXT *is comprised of the derivation rules in Figures 4 and 6, with the addition of rule R-Extract from Figure 5.*

Our extended calculus combines array reasoning, based on [10] and expressed by the rules in Figure 6, with the core rules of Figure 4 and the R-Extract rule. Unlike in BASE, those rules do not reduce nth and update. Instead, they reason about those operators directly and handle their combination with concatenation. Nth-Concat identifies the ith element of sequence y with the corresponding element selected from its normal form (see Lemma 4). Update-Concat operates similarly, applying update to all the components.

Update-Concat-Inv operates similarly on the updated sequence rather than on the original sequence. Nth-Unit captures the semantics of nth when applied to a unit term. Update-Unit is similar and distinguishes an update on an out-of-bounds index (different from 0) from an update within the bound. Nth-Intro is meant to ensure that Nth-Update (explained below) and Nth-Unit (explained above) are applicable whenever an update term exists in the constraints. Nth-Update captures the read-over-write axioms of arrays, adapted to consider their lengths (see, e.g., [10]). It distinguishes three cases: In the first, the update index is out of bounds. In the second, it is not out of bounds, and the corresponding nth term accesses the same index that was updated. In the third case, the index used in the nth term is different from the updated index. Update-Bound considers two cases: either the update changes the sequence, or the sequence remains the same. Finally, Nth-Split introduces a case split on the equality between two sequence variables x and x' whenever they appear as arguments to nth with equivalent second arguments. This is needed to ensure that we detect all cases where the arguments of two nth terms must be equal.

4.3 Correctness

In this section we prove the following theorem:

Theorem 1. *Let $X \in \{\mathsf{BASE}, \mathsf{EXT}\}$ and $(\mathsf{S}_0, \mathsf{A}_0)$ be a configuration, and assume without loss of generality that A_0 contains only arithmetic constraints that are not sequence constraints. Let T be a derivation tree obtained by applying the rules of X with $(\mathsf{S}_0, \mathsf{A}_0)$ as the initial configuration.*

1. *If T is closed, then $\mathsf{S}_0 \cup \mathsf{A}_0$ is T_{Seq}-unsatisfiable.*
2. *If T contains a saturated configuration (S, A) w.r.t. X, then (S, A) is T_{Seq}-satisfiable, and so is $(\mathsf{S}_0, \mathsf{A}_0)$.*

The theorem states that the calculi are correct in the following sense: if a closed derivation tree is obtained for the constraints $\mathsf{S}_0 \cup \mathsf{A}_0$ then those constraints are unsatisfiable in T_{Seq}; if a tree with a saturated leaf is obtained, then they are satisfiable. It is possible, however, that neither kind of tree can be derived by the calculi, making them neither refutation-complete nor terminating. This is not surprising since, as mentioned in the introduction, the decidability of even weaker theories is still unknown.

Proving the first claim in Theorem 1 reduces to a local soundness argument for each of the rules. For the second claim, we sketch below how to construct a satisfying model $\mathcal{M}$ from a saturated configuration for the case of EXT. The case for BASE is similar and simpler.

Model construction steps The full model construction and its correctness are described in a longer version of this paper [23] together with a proof of the theorem above. Here is a summary of the steps needed for the model construction.

1. Sorts: $\mathcal{M}(\mathsf{Elem})$ is interpreted as some arbitrary countably infinite set. $\mathcal{M}(\mathsf{Seq})$ and $\mathcal{M}(\mathsf{Int})$ are then determined by the theory.
2. Σ_{Seq}-symbols: T_{Seq} enforces the interpretation of almost all Σ_{Seq}-symbols, except for nth when the second input is out of bounds. We cover this case below.
3. Integer variables: based on the saturation of A-Conf, we know there is some T_{LIA}-model satisfying A. We set $\mathcal{M}$ to interpret integer variables according to this model.

$$\textsf{Nth-Concat}\ \frac{x \approx \mathsf{nth}(y,i) \in \mathsf{S} \qquad \mathsf{S} \models^{*}_{+\!\!+} y \approx w_1 +\!\!+ \cdots +\!\!+ w_n}{\begin{array}{c} \mathsf{A} := \mathsf{A}, i < 0 \vee i \geqslant \ell_y \quad \| \\ \mathsf{A} := \mathsf{A}, 0 \leqslant i < \ell_{w_1} \qquad \mathsf{S} := \mathsf{S}, x \approx \mathsf{nth}(w_1, i) \quad \| \quad \dots \quad \| \\ \mathsf{A} := \mathsf{A}, \sum_{j=1}^{n-1} \ell_{w_j} \leqslant i < \sum_{j=1}^{n} \ell_{w_j} \qquad \mathsf{S} := \mathsf{S}, x \approx \mathsf{nth}(w_n, i - \sum_{j=1}^{n-1} \ell_{w_j}) \end{array}}$$

$$\textsf{Update-Concat}\ \frac{x \approx \mathsf{update}(y,i,v) \in \mathsf{S} \qquad \mathsf{S} \models^{*}_{+\!\!+} y \approx w_1 +\!\!+ \cdots +\!\!+ w_n}{\begin{array}{c} \mathsf{S} := \mathsf{S}, x \approx z_1 +\!\!+ \cdots +\!\!+ z_n, \\ z_1 \approx \mathsf{update}(w_1, i, v), \dots, z_n \approx \mathsf{update}(w_n, i - \sum_{j=1}^{n-1} \ell_{w_j}, v) \end{array}}$$

$$\textsf{Update-Concat-Inv}\ \frac{x \approx \mathsf{update}(y,i,v) \in \mathsf{S} \qquad \mathsf{S} \models^{*}_{+\!\!+} x \approx w_1 +\!\!+ \cdots +\!\!+ w_n}{\begin{array}{c} \mathsf{S} := \mathsf{S}, y \approx z_1 +\!\!+ \cdots +\!\!+ z_n, \\ w_1 \approx \mathsf{update}(z_1, i, v), \dots, w_n \approx \mathsf{update}(z_n, i - \sum_{j=1}^{n-1} \ell_{w_j}, v) \end{array}}$$

$$\textsf{Nth-Unit}\ \frac{x \approx \mathsf{nth}(y,i) \in \mathsf{S} \qquad \mathsf{S} \models y \approx \mathsf{unit}(u)}{\mathsf{A} := \mathsf{A}, i < 0 \vee i > 0 \quad \| \quad \mathsf{A} := \mathsf{A}, i \approx 0 \qquad \mathsf{S} := \mathsf{S}, x \approx u}$$

$$\textsf{Update-Unit}\ \frac{x \approx \mathsf{update}(y,i,v) \in \mathsf{S} \qquad \mathsf{S} \models y \approx \mathsf{unit}(u)}{\begin{array}{c} \mathsf{A} := \mathsf{A}, i < 0 \vee i > 0 \qquad \mathsf{S} := \mathsf{S}, x \approx \mathsf{unit}(u) \quad \| \\ \mathsf{A} := \mathsf{A}, i \approx 0 \qquad \mathsf{S} := \mathsf{S}, x \approx \mathsf{unit}(v) \end{array}}$$

$$\textsf{Nth-Intro}\ \frac{s' \approx \mathsf{update}(s,i,t) \in \mathsf{S}}{\mathsf{S} := \mathsf{S}, e \approx \mathsf{nth}(s,i), e' \approx \mathsf{nth}(s',i)}$$

$$\textsf{Nth-Update}\ \frac{\mathsf{nth}(x,j) \in \mathcal{T}(\mathsf{S}) \qquad y \approx \mathsf{update}(z,i,v) \in \mathsf{S} \qquad \mathsf{S} \models x \approx y \text{ or } \mathsf{S} \models x \approx z}{\begin{array}{c} \mathsf{A} := \mathsf{A}, j < 0 \vee j \geqslant \ell_x \quad \| \\ \mathsf{A} := \mathsf{A}, i \approx j, 0 \leqslant j < \ell_x \qquad \mathsf{S} := \mathsf{S}, \mathsf{nth}(y,j) \approx v \quad \| \\ \mathsf{A} := \mathsf{A}, i \not\approx j, 0 \leqslant j < \ell_x \qquad \mathsf{S} := \mathsf{S}, \mathsf{nth}(y,j) \approx \mathsf{nth}(z,j) \end{array}}$$

$$\textsf{Update-Bound}\ \frac{x \approx \mathsf{update}(y,i,v) \in \mathsf{S}}{\mathsf{A} := \mathsf{A}, 0 \leqslant i < \ell_y \qquad \mathsf{S} := \mathsf{S}, \mathsf{nth}(y,i) \not\approx v \quad \| \quad \mathsf{S} := \mathsf{S}, x \approx y}$$

$$\textsf{Nth-Split}\ \frac{\mathsf{nth}(x,i), \mathsf{nth}(x',i') \in \mathcal{T}(\mathsf{S}) \qquad i \approx i' \in \mathsf{A}}{\mathsf{S} := \mathsf{S}, x \approx x' \quad \| \quad \mathsf{S} := \mathsf{S}, x \not\approx x'}$$

Fig. 6: Extended derivation rules. The rules use $z_1, \dots, z_n$ to denote fresh sequence variables and e, e' to denote fresh element variables.

4. Element variables: these are partitioned into their $\equiv_{\mathsf{S}}$ equivalence classes. Each class is assigned a distinct element from $\mathcal{M}(\mathsf{Elem})$, which is possible since it is infinite.
5. Atomic sequence variables: these are assigned interpretations in several sub-steps:
   (a) length: we first use the assignments to variables ℓ_x to set the length of $\mathcal{M}(x)$, without assigning its actual value.
   (b) unit variables: for variables x with $x \equiv_{\mathsf{S}} \mathsf{unit}(z)$, we set $\mathcal{M}(x)$ to be $[\mathcal{M}(z)]$.
   (c) non-unit variables: All other sequence variables are assigned values according to a *weak equivalence graph* we construct in a manner similar to [10]. This construction takes into account constraints that involve update and nth.
6. Non-atomic sequence variables: these are first transformed to their unique normal form (see Lemma 4), consisting of concatenations of atomic variables. Then, the values assigned to these variables are concatenated.

7. nth-terms: for out-of-bounds indices in nth-terms, we rely on $\equiv_{\mathsf{S}}$ to make sure that the assignment is consistent.

We conclude this section with an example of the construction of $\mathcal{M}$.

Example 5. Consider a signature in which Elem is Int, and a saturated configuration $(\mathsf{S}^*, \mathsf{A}^*)$ w.r.t. EXT that includes the following formulas: $y \approx y_1 ++ y_2$, $x \approx x_1 ++ x_2$, $y_2 \approx x_2$, $y_1 \approx \mathsf{update}(x_1, i, a)$, $|y_1| = |x_1|$, $|y_2| = |x_2|$, $\mathsf{nth}(y, i) \approx a$, $\mathsf{nth}(y_1, i) \approx a$. Following the above construction, a satisfying interpretation $\mathcal{M}$ can be built as follows:

Step 1 Set both $\mathcal{M}(\mathsf{Int})$ and $\mathcal{M}(\mathsf{Elem})$ to be the set of integer numbers. $\mathcal{M}(\mathsf{Seq})$ is fixed by the theory.

Step 3, Step 4 First, find an arithmetic model, $\mathcal{M}(\ell_x) = \mathcal{M}(\ell_y) = 4, \mathcal{M}(\ell_{y_1}) = \mathcal{M}(\ell_{x_1}) = 2, \mathcal{M}(\ell_{y_2}) = \mathcal{M}(\ell_{x_2}) = 2, \mathcal{M}(i) = 0$. Further, set $\mathcal{M}(a) = 0$.

Step 5a Start assigning values to sequences. First, set the lengths of $\mathcal{M}(x)$ and $\mathcal{M}(y)$ to be 4, and the lengths of $\mathcal{M}(x_1), \mathcal{M}(x_2), \mathcal{M}(y_1), \mathcal{M}(y_2)$ to be 2.

Step 5b is skipped as there are no unit terms.

Step 5c Set the 0th element of $\mathcal{M}(y_1)$ to 0 to satisfy $\mathsf{nth}(y_1, i) = a$ (y_1 is atomic, y is not). Assign fresh values to the remaining indices of atomic variables. The result can be, e.g., $\mathcal{M}(y_1) = [0, 2], \mathcal{M}(x_1) = [1, 2], \mathcal{M}(y_2) = \mathcal{M}(x_2) = [3, 4]$.

Step 6 Assign non-atomic sequence variables based on equivalent concatenations: $\mathcal{M}(y) = [0, 2, 3, 4], \mathcal{M}(x) = [1, 2, 3, 4]$.

Step 7 No integer variable in the formula was assigned an out-of-bound value, and so the interpretation of nth on out-of-bounds cases is set arbitrarily.

5 Implementation

We implemented our procedure for sequences as an extension of a previous theory solver for strings [17, 22]. This solver is integrated in cvc5, and has been generalized to reason about both strings and sequences. In this section, we describe how the rules of the calculus are implemented and the overall strategy for when they are applied.

Like most SMT solvers, cvc5 is based on the CDCL(T) architecture [19] which combines several subsolvers, each specialized on a specific theory, with a solver for propositional satisfiability (SAT). Following that architecture, cvc5 maintains an evolving set of formulas F. When F starts with quantifier-free formulas over the theory T_{Seq}, the case targeted by this work, the SAT solver searches for a satisfying assignment for F, represented as the set M of literals it satisfies. If none exists, the problem is unsatisfiable at the propositional level and hence T_{Seq}-unsatisfiable. Otherwise, M is partitioned into the arithmetic constraints A and the sequence constraints S and checked for T_{Seq}-satisfiability using the rules of the EXT calculus. Many of those rules, including all those with multiple conclusions, are implemented by adding new formulas to F (following the splitting-on-demand approach [4]). This causes the SAT solver to try to extend its assignment to those formulas, which results in the addition of new literals to M (and thereby also to A and S).

In this setting, the rules of the two calculi are implemented as follows. The effect of rule $\mathsf{A\text{-}Conf}$ is achieved by invoking cvc5's theory solver for linear integer arithmetic. Rule $\mathsf{S\text{-}Conf}$ is implemented by the congruence closure submodule of the theory solver for sequences. Rules A-Prop and S-Prop are implemented by the standard mechanism for theory combination. Note that each of these four rules may be applied *eagerly*, that is, before constructing a complete satisfying assignment M for F.

The remaining rules are implemented in the theory solver for sequences. Each time M is checked for satisfiability, cvc5 follows a strategy to determine which rule to apply next. If none of the rules apply and the configuration is different from unsat, then it is saturated, and the solver returns sat. The strategy for EXT prioritizes rules as follows. Only the first applicable rule is applied (and then control goes back to the SAT solver).

1. (Add length constraints) For each sequence term in S, apply L-Intro or L-Valid, if not already done. We apply L-Intro for non-variables, and L-Valid for variables.
2. (Mark congruent terms) For each set of update (resp. nth) terms that are congruent to one another in the current configuration, mark all but one term and ignore the marked terms in the subsequent steps.
3. (Reduce extract) For extract(y, i, j) in S, apply R-Extract if not already done.
4. (Construct normal forms) Apply U-Eq or C-Split. We choose how to apply the latter rule based on constructing normal forms for equivalence classes in a bottom-up fashion, where the equivalence classes of x and y are considered before the equivalence class of $x +\!\!+ y$. We do this until we find an equivalence class such that S $\models^*_{++} z \approx u_1$ and S $\models^*_{++} z \approx u_2$ for distinct u_1, u_2.
5. (Normal forms) Apply C-Eq if two equivalence classes have the same normal form.
6. (Extensionality) For each disequality in S, apply Deq-Ext, if not already done.
7. (Distribute update and nth) For each term update(x, i, t) (resp. nth(x, j)) such that the normal form of x is a concatenation term, apply Update-Concat and Update-Concat-Inv (resp. Nth-Concat) if not already done. Alternatively, if the normal form of the equivalence class of x is a unit term, apply Update-Unit (resp. Nth-Unit).
8. (Array reasoning on atomic sequences) Apply Nth-Intro and Update-Bound to update terms. For each update term, find the matching nth terms and apply Nth-Update. Apply Nth-Split to pairs of nth terms with equivalent indices.
9. (Theory combination) Apply S-A for all arithmetic terms occurring in both S and A.

Whenever a rule is applied, the strategy will restart from the beginning in the next iteration. The strategy is designed to apply with higher priority steps that are easy to compute and are likely to lead to conflicts. Some steps are ordered based on dependencies from other steps. For instance, Steps 5 and 7 use normal forms, which are computed in Step 4. The strategy for the BASE calculus is the same, except that Steps 7 and 8 are replaced by one that applies R-Update and R-Nth to all update and nth terms in S.

We point out that the C-Split rule may cause non-termination of the proof strategy described above in the presence of *cyclic* sequence constraints, for instance, constraints where sequence variables appear on both sides of an equality. The solver uses methods for detecting some of these cycles, to restrict when C-Split is applied. In particular, when S $\models^*_{++} x \approx (\overline{\boldsymbol{u}} +\!\!+ s +\!\!+ \overline{\boldsymbol{w}})\!\downarrow$, S $\models^*_{++} x \approx (\overline{\boldsymbol{u}} +\!\!+ t +\!\!+ \overline{\boldsymbol{v}})\!\downarrow$, and s occurs in $\overline{\boldsymbol{v}}$, then C-Split is not applied. Instead, other heuristics are used, and in some cases the solver terminates with a response of “unknown” (see e.g., [17] for details). In addition to the version shown here, we also use another variation of the C-Split rule where the normal forms are matched in reverse (starting from the last terms in the concatenations). The implementation also uses fast entailment tests for length inequalities. These tests may allow us to conclude which branch of C-Split, if any, is feasible, without having to branch on cases explicitly.

Although not shown here, the calculus can also accommodate certain *extended* sequence constraints, that is, constraints using a signature with additional functions. For example, our implementation supports sequence containment, replacement, and reverse. It also supports an extended variant of the update operator, in which the third argument is a sequence that overrides the sequence being updated starting from the index given in the second argument. Constraints involving these functions are handled by reduction rules, similar to those shown in Figure 5. The implementation is further optimized by using context-dependent simplifications, which may eagerly infer when certain sequence terms can be simplified to constants based on the current set of assertions [22].

6 Evaluation

We evaluate the performance of our approach, as implemented in cvc5. The evaluation investigates: (i) whether the use of sequences is a viable option for reasoning about vectors in programs, (ii) how our approach compares with other sequence solvers, and (iii) what is the performance impact of our array-style extended rules. As a baseline, we use Version 4.8.14 of the Z3 SMT solver, which supports a theory of sequences without updates. For cvc5, we evaluate implementations of both the basic calculus (denoted **cvc5**) and the extended array-based calculus (denoted **cvc5-a**). The benchmarks, solver configurations, and logs from our runs are available for download.[4] We ran all experiments on a cluster equipped with Intel Xeon E5-2620 v4 CPUs. We allocated one physical CPU core and 8GB of RAM for each solver-benchmark pair and used a time limit of 300 seconds. We use the following two sets of benchmarks:

Array Benchmarks (ARRAYS) The first set of benchmarks is derived from the QF_AX benchmarks in SMT-LIB [3]. To generate these benchmarks, we (i) replace declarations of arrays with declarations of sequences of uninterpreted sorts, (ii) change the sort of index terms to integers, and (iii) replace store with update and select with nth. The resulting benchmarks are quantifier-free and do not contain concatenations. Note that the original and the derived benchmarks are not equisatisfiable, because sequences take into account out-of-bounds cases that do not occur in arrays. For the Z3 runs, we add to the benchmarks a definition of update in terms of extraction and concatenation.

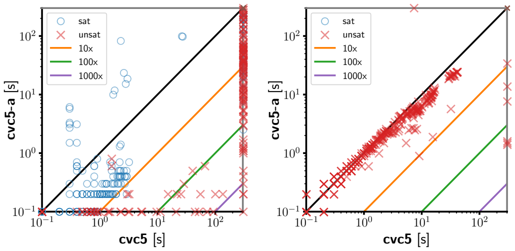
Smart Contract Verification (DIEM) The second set of benchmarks consists of verification conditions generated by running the Move Prover [24] on smart contracts written for the Diem framework. By default, the encoding does not use the sequence update operation, and so Z3 can be used directly. However, we also modified the Move Prover encoding to generate benchmarks that do use the update operator, and ran cvc5 on them. In addition to using the sequence theory, the benchmarks make heavy use of quantifiers and the SMT-LIB theory of datatypes.

Figure 7a summarizes the results in terms of number of solved benchmarks and total time in seconds on commonly solved benchmarks. The configuration that solves the

[4] http://dx.doi.org/10.5281/zenodo.6146565

		w/ update		
Set		**cvc5**	**cvc5-a**	**Z3**
Arrays	Slvd	242	**390**	170
(551)	Time	162	303	4329
Diem	Slvd	542	**547**	443
(558)	Time	518	440	639

(a)



(b) Arrays

(c) Diem

Fig. 7: Figure 7a lists the number of solved benchmarks and total time on commonly solved benchmarks. The scatter plots compare the base solver (**cvc5**) and the extended solver (**cvc5-a**) on Array (Figure 7b) and Diem (Figure 7c) benchmarks.

largest number of benchmarks is the implementation of the extended calculus (**cvc5-a**). This approach also successfully solves most of the Diem benchmarks, which suggests that sequences are a promising option for encoding vectors in programs. The results further show that the sequences solver of cvc5 significantly outperforms Z3 on both the number of solved benchmarks and the solving time on commonly solved benchmarks.

Figures 7b and 7c show scatter plots comparing **cvc5** and **cvc5-a** on the two benchmark sets. We can see a clear trend towards better performance when using the extended solver. In particular, the table shows that in addition to solving the most benchmarks, **cvc5-a** is also fastest on the commonly solved instances from the Diem benchmark set.

For the Arrays set, we can see that some benchmarks are slower with the extended solver. This is also reflected in the table, where **cvc5-a** is slower on the commonly solved instances. This is not too surprising, as the extra machinery of the extended solver can sometimes slow down easy problems. As problems get harder, however, the benefit of the extended solver becomes clear. For example, if we drop Z3 and consider just the commonly solved instances between **cvc5** and **cvc5-a** (of which there are 242), **cvc5-a** is about 2.47× faster (426 vs 1053 seconds). Of course, further improving the performance of **cvc5-a** is something we plan to explore in future work.

7 Conclusion

We introduced calculi for checking satisfiability in the theory of sequences, which can be used to model the vector data type. We described our implementation in cvc5 and provided an evaluation, showing that the proposed theory is rich enough to naturally express verification conditions without introducing quantifiers, and that our implementation is efficient. We believe that verification tools can benefit by changing their encoding of verification conditions that involve vectors to use the proposed theory and implementation.

We plan to propose the incorporation of this theory in the SMT-LIB standard and contribute our benchmarks to SMT-LIB. As future research, we plan to integrate other approaches for array solving into our basic solver. We also plan to study the politeness [16, 20] and decidability of various fragments of the theory of sequences.

References

[1] F. Alberti, S. Ghilardi, and E. Pagani. Cardinality constraints for arrays (decidability results and applications). *Formal Methods Syst. Des.*, 51(3):545–574, 2017.

[2] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[3] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[4] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[5] C. W. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[6] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In D. Stewart and G. Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.

[7] N. Bjørner, L. de Moura, L. Nachmanson, and C. Wintersteiger. Programming Z3. `https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-sequences-and-strings`, 2018.

[8] N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An SMT-LIB format for sequences and regular expressions. *SMT*, 12:76–86, 2012.

[9] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2009.

[10] J. Christ and J. Hoenicke. Weakly equivalent arrays. In *FroCos*, volume 9322 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2015.

[11] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[12] N. Elad, S. Rain, N. Immerman, L. Kovács, and M. Sagiv. Summing up smart transitions. In *CAV (1)*, volume 12759 of *Lecture Notes in Computer Science*, pages 317–340. Springer, 2021.

[13] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 2 edition, 2001.

[14] S. Falke, F. Merz, and C. Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *VSTTE*, volume 8164 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2013.

[15] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2012.

[16] D. Jovanovic and C. W. Barrett. Polite theories revisited. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2010.

[17] T. Liang, A. Reynolds, C. Tinelli, C. W. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.

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

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[20] S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2005. Extended technical report is available at `https://hal.inria.fr/inria-00070335/`.

[21] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli. Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 225–235. IEEE, 2020.

[22] A. Reynolds, M. Woo, C. W. Barrett, D. Brumley, T. Liang, and C. Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2017.

[23] Y. Sheng, A. Nötzli, A. Reynolds, Y. Zohar, D. Dill, W. Grieskamp, J. Park, S. Qadeer, C. Barrett, and C. Tinelli. Reasoning about vectors using an SMT theory of sequences. *CoRR*, 10.48550/ARXIV.2205.08095, 2022.

[24] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill. The Move prover. In S. K. Lahiri and C. Wang, editors, *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV '20)*, volume 12224 of *Lecture Notes in Computer Science*, pages 137–150. Springer International Publishing, July 2020.