Acta Informatica (2025) 62:33
https://doi.org/10.1007/s00236-025-00496-w

RESEARCH

Reasoning over n-indexed sequences in SMT

Hichem Rami Ait-El-Hara[1,2] · **François Bobot**[2] · **Guillaume Bury**[1]

Received: 2 December 2024 / Accepted: 4 July 2025 / Published online: 21 August 2025

Abstract
The SMT (Satisfiability Modulo Theories) theory of arrays is well-established and widely used, with various decision procedures and extensions developed for it. However, recent contributions suggest that developing tailored reasoning for some theories, such as sequences and strings, can be more efficient than reasoning over them through axiomatization over the theory of arrays. In this paper, we are interested in reasoning over n-indexed sequences as they are found in some programming languages, such as Ada. We propose an SMT theory of n-indexed sequences and explore different ways to represent and reason over n-indexed sequences using existing theories, as well as tailored calculi for this theory.

1 Introduction

In the SMT theory of sequences, sequences are viewed as a generalization of strings to non-character elements, with possibly infinite alphabets. These sequences are dynamically sized, and their theory has a rich signature, enabling operations such as selecting elements of a sequence by their index, concatenating sequences, extracting sub-sequences, and more. This expressiveness makes the theory of sequences well-suited for representing many common data structures in programming languages, such as arrays in the C language, lists in Python, etc.

In contrast, the theory of arrays is less expressive. It only supports selecting and updating a single value at a single index, and arrays have fixed sizes determined by the cardinal-

✉ Hichem Rami Ait-El-Hara
hichem.ait-el-hara@ocamlpro.com

François Bobot
francois.bobot@cea.fr

Guillaume Bury
guillaume.bury@ocamlpro.com

[1] OCamlPro, 75014 Paris, France

[2] Université Paris-Saclay, CEA, List, F-91120 Palaiseau, France

ity of the sort of indices. While sequences have dynamic lengths and allow operations on sub-sequences. To represent sequences using the theory of arrays, it is necessary to extend the theory and axiomatize the necessary properties, such as dynamic length and additional operations like concatenation and sub-sequence extraction.

We are interested in a variant of the theory of sequences, which we call the theory of n -indexed sequences. These n-indexed sequences are defined as ordered collections of values of the same sort indexed from a first index *n* to a last index *m*. Such sequences are present in some programming languages such as Ada. Since there is no dedicated theory for such sequences, reasoning over them cannot be achieved straightforwardly using the existing theories of arrays and sequences. It is therefore necessary to use extensions and axiomatizations to reason over them.

In this paper, we will present the theory of n-indexed sequences, its signature and semantics, as well as different approaches to reason over it. These include leveraging existing theories and adapting calculi designed for the theory of sequences to the theory of n-indexed sequences.

1.1 Related work

The SMT theory of sequences was introduced by Bjørner et al. [6]. Several works have since explored its syntax and semantics [2, 6] and its decidability [10, 12]. Our contribution builds upon a previously published extended abstract [1], in which we introduced the theory of n-indexed sequences and presented various reasoning approaches.

Our work draws on the contribution by Sheng et al. [17], which describes the calculi for the theory of sequences implemented in the cvc5 SMT solver [3]. The Z3 SMT solver [16] also supports the theory of sequences, although we are unaware of any published contributions detailing its reasoning techniques. The calculi for the theory of sequences described in [17] are based on calculi developed for the theories of strings [5, 14] and arrays [9], which have been generalized and adapted for sequence reasoning.

Other contributions took a different approach, relying more on the theory of arrays and extending it with properties present and desired in sequences, such as length constraints [7, 8, 11] and operations like concatenation [19].

2 Notation

In the rest of the paper, we refer to the theory of arrays as the Array theory, the theory of sequences as the Seq theory, the theory of n-indexed sequences as the NSeq theory and the theory of Algebraic Data Types as the ADT theory. We also use the following notation: $=$ for equality, $\equiv$ for logical equivalence, and $\implies$ for implication. If a formula $a = b$ is true in the context of the solver, we say that *a* and *b* are (semantically) equivalent.

We represent the sort of integers with Int, the sort of n-indexed sequences with NSeq E, where E is the sort of the elements stored in the n-indexed sequences. We refer to n-indexed sequences as n-sequences.

We use s and s_n, k_n, w_n, y_n, and z_n (where *n* is an integer) to represent n-sequence terms. The symbols *i* and *j* represent general index terms, *f* and *l* represent index terms that denote bounds of n-sequences, and *v* and *u* represent n-sequence element terms.

We present the calculi we developed as a set of inference rules that handle the symbols of the NSeq theory. In the inference rules, the statements above the line are the premises, and the statements below it are the conclusions. The|| symbol separates the different cases of the conclusions. Terms that do not appear in the premise of an inference rule but do appear in the conclusion are to be considered fresh variables.

3 The theory of n-indexed sequences

We present in this section the theory of n-indexed sequences. The signature of the NSeq theory is presented in Table 1, along with the notation of the symbols of the theory that we use in the remainder of the paper.

Definition 1 (*Bounds*) The bounds of an n-sequence s are its first and last indices, which are respectively denoted as f_s and l_s, and correspond to the values returned by the functions $\mathrm{nseq.first}(s)$ and $\mathrm{nseq.last}(s)$, respectively. An index i is said to be within the bounds of an n-sequence s if:

$$\mathrm{f}_s \leq i \leq \mathrm{l}_s$$

Definition 2 An n-sequence s is said to be empty if $\mathrm{l}_s < \mathrm{f}_s$. Two empty n-sequences s_1 and s_2 are equal if $\mathrm{f}_{s_1} = \mathrm{f}_{s_2}$ and $\mathrm{l}_{s_1} = \mathrm{l}_{s_2}$. Otherwise, they are distinct.

The following list describes the semantics of each symbol in the theory:

- f_s: the first index of s.
- l_s: the last index of s.
- $\mathrm{get}\,(s_1, i)$: If $\mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1}$, returns the element associated with i in s; otherwise, returns an uninterpreted value. An uninterpreted value is one that is not constrained and can be any value of the right sort.
- $\mathrm{set}\,(s_1, i, v)$: If $\mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1}$, creates a new n-sequence s_2 that has the same bounds as s_1, where $\forall k.\ \mathrm{f}_{s_1} \leq k \leq \mathrm{l}_{s_1} \implies \mathrm{get}\,(s_2, k) = \mathrm{ite}\,(k = i, v, \mathrm{get}\,(s_1, k))$. Otherwise, returns s_1.
- $\mathrm{const}\,(f, l, v)$: Creates an n-sequence s with $\mathrm{f}_s = f \wedge \mathrm{l}_s = l$, where $\forall k.\ f \leq k \leq l \implies \mathrm{get}\,(s, k) = v$.

Table 1 The signature of the theory of n-indexed sequences

SMT-LIB symbol	Sort	Notation
nseq.first	NSeq E → Int	$\mathrm{f}\_$
nseq.last	NSeq E → Int	$\mathrm{l}\_$
nseq.get	NSeq E → Int → E	$\mathrm{get}\,(\_, \_)$
nseq.set	NSeq E → Int → E → NSeq E	$\mathrm{set}\,(\_, \_, \_)$
nseq.const	Int → Int → E → NSeq E	$\mathrm{const}\,(\_, \_, \_)$
nseq.relocate	NSeq E → Int → NSeq E	$\mathrm{relocate}\,(\_, \_)$
nseq.concat	NSeq E → NSeq E → NSeq E	$\mathrm{concat}\,(\_, \_)$
nseq.slice	NSeq E → Int → Int → NSeq E	$\mathrm{slice}\,(\_, \_, \_)$
nseq.update	NSeq E → NSeq E → NSeq E	$\mathrm{update}\,(\_, \_)$

- $\mathrm{relocate}\,(s_1, f)$: Given an n-sequence s_1 and an index f, returns a new n-sequence s_2 with $\mathrm{f}_{s_2} = f \wedge \mathrm{l}_{s_2} = f + \mathrm{l}_{s_1} - \mathrm{f}_{s_1}$, where $\forall k.\ f \leq k \leq f + \mathrm{l}_{s_1} - \mathrm{f}_{s_1} \implies \mathrm{get}\,(s_2, k) = \mathrm{get}\,(s_1, k - \mathrm{f}_{s_2} + \mathrm{f}_{s_1})$.
- $\mathrm{concat}\,(s_1, s_2)$: If s_1 is empty, returns s_2. If s_2 is empty, returns s_1. If $\mathrm{f}_{s_2} = \mathrm{l}_{s_1} + 1$, returns a new n-sequence s_3 with $\mathrm{f}_{s_3} = \mathrm{f}_{s_1} \wedge \mathrm{l}_{s_3} = \mathrm{l}_{s_2}$, where $\forall k.\ \mathrm{f}_{s_1} \leq k \leq \mathrm{l}_{s_2} \implies \mathrm{get}\,(s_3, k) = \mathrm{ite}\,(k \leq \mathrm{l}_{s_1}, \mathrm{get}\,(s_1, k), \mathrm{get}\,(s_2, k))$. Otherwise, returns s_1.
- $\mathrm{slice}\,(s_1, f, l)$: If $\mathrm{f}_{s_1} \leq f \leq l \leq \mathrm{l}_{s_1}$, returns a new n-sequence s_2 with $\mathrm{f}_{s_2} = f \wedge \mathrm{l}_{s_2} = l$, where $\forall k.\ f \leq k \leq l \implies \mathrm{get}\,(s_2, k) = \mathrm{get}\,(s_1, k)$. Otherwise, returns s.
- $\mathrm{update}\,(s_1, s_2)$: If s_1 is empty, s_2 is empty, or the property $\mathrm{f}_{s_1} \leq \mathrm{f}_{s_2} \leq \mathrm{l}_{s_2} \leq \mathrm{l}_{s_1}$ does not hold, returns s_1. Otherwise, returns a new n-sequence s_3 that has the same bounds as s_1, where $\forall k.\ \mathrm{f}_{s_1} \leq k \leq \mathrm{l}_{s_1} \implies \mathrm{get}\,(s_3, k) = \mathrm{ite}\,(\mathrm{f}_{s_2} \leq k \leq \mathrm{l}_{s_2}, \mathrm{get}\,s_2, k, \mathrm{get}\,s_1, k)$.

Definition 3 (*Extensionality*) The theory of n-indexed sequences is extensional, which means that n-sequences that have the same bounds and contain the same elements are equal. Therefore, given two n-sequences s_1 and s_2:

$$s_1 = s_2 \equiv$$
$$\mathrm{f}_{s_1} = \mathrm{f}_{s_2} \wedge \mathrm{l}_{s_1} = \mathrm{l}_{s_2} \wedge (\forall i.\ \mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1} \rightarrow \mathrm{get}\,(s_1, i) = \mathrm{get}\,(s_2, i))$$

Different semantics can be chosen for the functions of this theory, particularly the slice and update functions. In [2], we defined a set of theory design criteria. In particular, we showed that previously proposed semantics for the update function in the Seq theory were not symmetric (an overlapping update on the right was different from one on the left), which does not align with the design criterion of avoiding surprising the users. Instead, we proposed a symmetric semantic: in all cases of overlapping update, the shared indices are updated. It is possible to adopt the same semantics for the update operator in the NSeq theory. However, we chose a different, yet still symmetrical, semantic by not updating the n-sequence whenever the update overlaps its bounds. This choice is justified by our main use case, which is representing arrays from the Ada programming language. That is also the reason for the choice of the semantics of the concat and slice functions.

4 Reasoning with existing theories

One way to reason over the NSeq theory is by using the theory of arrays. It is done by extending it with the symbols of the NSeq theory and adding the right axioms that capture the semantics of the corresponding symbols in the NSeq theory. However, this approach has considerable limitations, as operations on slices of n-sequences are handled using axioms that quantify over all the elements of those slices, and the handling of these quantifiers tends to be costly for solvers.

Alternatively, it is possible to use the Seq and ADT theories to encode n-sequences. This can be done by defining n-sequences as a pair of a sequence and the first index (the offset to zero):

```
(declare-datatype NSeq (par (T)
  ((nseq.mk (nseq.first Int) (nseq.seq (Seq T))))))
```

The other symbols of the NSeq theory can also be defined using the NSeq data type defined above, for example:

```
(define-fun nseq.last (par (T) ((s (NSeq T))) Int
  (+ (- (seq.len (nseq.seq s)) 1) (nseq.first s))))

(define-fun nseq.get (par (T) ((s (NSeq T)) (i Int)) T
  (seq.nth (nseq.seq s) (- i (nseq.first s)))))

(define-fun nseq.set (par (T)
  ((s (NSeq T)) (i Int) (v T)) (NSeq T)
    (nseq.mk (nseq.first s)
      (seq.update
        (nseq.seq s) (- i (nseq.first s)) (seq.unit v)))))
```

Except for the const function which needs to be axiomatized:

```
(declare-fun nseq.const (par (T) (Int Int T) (NSeq T)))

;; "nseq_const"
(assert (par (T) (forall ((f Int) (l Int) (v T))
  (!
    (let ((s (nseq.const f l v)))
      (and
        (= (nseq.first s) f)
        (= (nseq.last s) l)
        (forall ((i Int)) (=>
          (and (<= f i) (<= i l))
          (= (nseq.get s i) v)))))
  :pattern ((nseq.const f l v))))))
```

The full NSeq theory, defined using the Seq and ADT theories, is included in Appendix A.

Although this approach allows us to reason over n-indexed sequences, it is not ideal to depend on two theories to do so, as it implies that the performance of reasoning about the theory is tied to the performance of reasoning over the other two theories. Additionally, the differences in semantics between the update and slice functions of the NSeq theory and the seq.update and seq.extract functions of the Seq theory make the definitions relatively complex and costly to handle by solvers.

Another difference is in empty n-sequences. With this encoding, empty n-sequences that have the same first index will always have the same last index (one subtracted from the first index). On the other hand, in the original theory, an empty n-sequence can have any last index that is lesser than the first index. The encoding could be made faithful to the original theory by adding a specific constructor in the ADT for empty n-sequences. However, this would eventually hinder the performance of the solvers that use the encoding, especially since the difference in semantics is not problematic. Empty n-sequences typically represent corner cases or cases of failure, and the position of their last index tends to be irrelevant for determining satisfiability.

5 Porting calculi from the Seq theory to the NSeq theory

To develop our calculi over the NSeq theory, we based our work on the calculi developed by Sheng et al. [17] for the Seq theory, where two calculi were proposed. The first is called the BASE calculus, which is based on a string theory calculus that reduces the functions of the theory to concatenations of sequences. The second is called the EXT calculus, which handles the functions of the theory that select and store an element at an index using array-like reasoning. Our versions of these calculi are referred to as NS-BASE and NS-EXT, respectively.

The NSeq theory differs from the Seq theory in both the syntax and semantics of many symbols:

- const and $\mathrm{relocate}$ do not appear in the Seq theory, while seq.empty, seq.unit, and seq.len do not appear in the NSeq theory.
- The seq.nth function corresponds to the get function in the NSeq theory.
- seq.update from the Seq theory, with a value as the third argument, corresponds to set in the NSeq theory, while seq.update with a sequence as the third argument corresponds to update in the NSeq theory, which takes only two n-sequences as arguments.
- seq.extract in the Seq theory takes a sequence, an offset, and a length, and corresponds to slice in the NSeq theory, which takes an n-sequence, a first index, and a last index.
- The concatenation function (seq.++) in the Seq theory is n-ary, and it corresponds to concat in the NSeq theory, which is binary.

Therefore, we needed to make substantial changes to the Seq theory calculi to adapt them to the NSeq theory. In this section, we present the resulting calculi. We assume that we are in a theory combination framework where reasoning with the theories of integer arithmetic and booleans is supported, and where unsatisfiability in one of the theories implies unsatisfiability of the entire reasoning.

5.1 Common calculus

Definition 4 (*Equivalence modulo relocation*) Given two n-sequences s_1 and s_2, the terms are said to be equivalent modulo relocation, denoted by the relation $s_1 =_{reloc} s_2$, which is defined as:

$$s_1 =_{reloc} s_2 \equiv$$
$$\mathrm{l}_{s_2} = \mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2} \wedge \forall i : Int,\, \mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1} \Rightarrow \mathrm{get}\,(s_1, i) = \mathrm{get}\,(s_2, i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2})$$

Two n-sequences are equivalent modulo relocation if they are equal or start at different indices but contain the same sequence of elements.

Proposition 1 The equivalence modulo relocation relation is an equivalence relation between n-sequences.

Proof The proof of Proposition 1 is in Appendix B. □

Definition 5 (n-sequence *normal form*) For simplicity and consistency with the Seq theory calculi, we introduce an internal concatenation operator ::, for which the following invariant holds:

$$s = s_1 :: s_2 \quad \Longrightarrow \quad \mathrm{f}_s = \mathrm{f}_{s_1} \wedge \mathrm{l}_s = \mathrm{l}_{s_2} \wedge \mathrm{f}_{s_2} = \mathrm{l}_{s_1} + 1$$

This operator is used to normalize n-sequences. It differs from concat in that it does not require checking the condition $\mathrm{f}_{s_2} = \mathrm{l}_{s_1} + 1$ before concatenation, as this condition is ensured by the invariant.

Assumption 1 We assume that the following simplification rewrites are applied whenever possible:

$$\begin{array}{lcll} s_1 :: s_2 & \rightarrow & s_1 & \text{when } \mathrm{l}_{s_2} < \mathrm{f}_{s_2} \quad (1) \\ s_1 :: s_2 & \rightarrow & s_2 & \text{when } \mathrm{l}_{s_1} < \mathrm{f}_{s_1} \quad (2) \\ s_1 :: s_2 & \rightarrow & s_1 :: w_1 :: \ldots :: w_n & \text{when } s_2 = w_1 :: \ldots :: w_n \quad (3) \\ s_1 :: s_2 & \rightarrow & w_1 :: \ldots :: w_n :: s_2 & \text{when } s_1 = w_1 :: \ldots :: w_n \quad (4) \end{array}$$

(1) and (2) remove empty n-sequences from the normal form. (3) and (4) ensure that when an n-sequence appears in the normal form of another one and has its own normal form, then it is replaced by its normal form.

Figure 1 illustrates a set of common rules shared between the two calculi NS-BASE and NS-EXT. The rules Const-Bounds and Reloc-Bounds propagate the bounds of constant and relocated n-sequences, which are created using the const and $\mathrm{relocate}$ functions, respectively. The rules NS-Slice, NS-Concat, and NS-Update handle slice, concat, and update by normalizing the n-sequences under appropriate conditions.

If an n-sequence has two normal forms where distinct terms begin at the same index but end at different indices, the NS-Split rule rewrites the longer term as a concatenation of the shorter one and a fresh variable. The NS-Comp-Reloc rule propagates concatenations over the $=_{reloc}$ rule, while Reloc-Inv ensures that two n-sequences that are equivalent modulo relocation are equal if they start at the same index. The NS-Exten rule is the extensionality rule, which states that any two n-sequences s_1 and s_2 are either equal or distinct. They can be distinct either for having different bounds, for containing distinct components that have the same bounds, or differing in at least one element.

5.2 The base calculus

The base calculus comprises the rules in Figures 1 and 2. The rules R-Get and R-Set handle the get and set operations by introducing new normal forms for the n-sequences they operate on. In the R-Get rule, when *i* is within the bounds of *s*, a new normal form of *s* is introduced. This form includes a constant n-sequence of size one at the *i*th position storing the value *v*, and two variables, k_1 and k_2, to represent the left and right segments of the n -sequence *s*, respectively.

The R-Set rule operates similarly: when *i* is within the bounds of s_2, new normal forms are introduced for both s_1 and s_2. These forms share two variables, k_1 and k_3, which represent the segments to the left and right of the *i*th index. For s_1, the normal form contains a constant n-sequence of size one holding the value *v* at the *i*th index, while s_2's normal form contains an n-sequence variable, k_2, also of size one at the *i*th index.

$$\textsf{Const-Bounds}\ \frac{s = \text{const}(f, l, v)}{\text{f}_s = f \wedge \text{l}_s = l} \qquad \textsf{Reloc-Bounds}\ \frac{s_1 = \text{relocate}(s_2, i)}{\begin{array}{l} i = \text{f}_{s_2} \wedge s_1 = s_2 \quad || \\ i \neq \text{f}_{s_2} \wedge \text{f}_{s_1} = i \wedge \text{l}_{s_1} = i + \text{l}_{s_2} - \text{f}_{s_2} \\ \wedge s_1 =_{reloc} s_2 \end{array}}$$

$$\textsf{NS-Slice}\ \frac{s_1 = \text{slice}(s, f, l)}{\begin{array}{c} (f < \text{f}_s \vee l < f \vee \text{l}_s < l) \wedge s_1 = s \quad || \\ \text{f}_s \leq f \leq l \leq \text{l}_s \wedge \\ \text{f}_{s_1} = f \wedge \text{l}_{s_1} = l \wedge s = k_1 :: s_1 :: k_2 \end{array}} \qquad \textsf{Reloc-Inv}\ \frac{s_1 =_{reloc} s_2}{\begin{array}{l} \text{f}_{s_1} = \text{f}_{s_2} \wedge s_1 = s_2 \ || \\ \text{f}_{s_1} \neq \text{f}_{s_2} \wedge s_1 \neq s_2 \end{array}}$$

$$\textsf{NS-Concat}\ \frac{s = \text{concat}(s_1, s_2)}{\begin{array}{l} \text{l}_{s_1} < \text{f}_{s_1} \wedge s = s_2 \quad || \\ (\text{l}_{s_2} < \text{f}_{s_2} \vee \text{l}_{s_1} + 1 \neq \text{f}_{s_2}) \wedge s = s_1 \quad || \\ \text{f}_{s_1} \leq \text{l}_{s_1} \wedge \text{f}_{s_2} \leq \text{l}_{s_2} \wedge \text{f}_{s_2} = \text{l}_{s_1} + 1 \wedge s = s_1 :: s_2 \end{array}}$$

$$\textsf{NS-Update}\ \frac{s_1 = \text{update}(s_2, s)}{\begin{array}{l} (\text{l}_s < \text{f}_s \vee f_s < \text{f}_{s_2} \vee \text{l}_{s_2} < \text{l}_s) \wedge s_1 = s_2 \quad || \\ \text{f}_{s_2} \leq \text{f}_s \leq \text{l}_s \leq \text{l}_{s_2} \wedge s_1 = k_1 :: s :: k_3 \wedge s_2 = k_1 :: k_2 :: k_3 \end{array}}$$

$$\textsf{NS-Split}\ \frac{s = w :: y_1 :: z_1 \qquad s = w :: y_2 :: z_2}{\begin{array}{l} \text{l}_{y_1} = \text{l}_{y_2} \wedge y_1 = y_2 \quad || \\ \text{l}_{y_1} > \text{l}_{y_2} \wedge y_1 = y_2 :: k \wedge \text{f}_k = \text{l}_{y_2} + 1 \wedge \text{l}_k = \text{l}_{y_1} \ || \\ \text{l}_{y_1} < \text{l}_{y_2} \wedge y_2 = y_1 :: k \wedge \text{f}_k = \text{l}_{y_1} + 1 \wedge \text{l}_k = \text{l}_{y_2} \end{array}}$$

$$\textsf{NS-Comp-Reloc}\ \frac{s_1 = k_1 :: k_2 :: \ldots :: k_n \qquad s_1 =_{reloc} s_2}{\begin{array}{l} \text{f}_{s_1} = \text{f}_{s_2} \wedge s_1 = s_2 \quad || \\ s_2 = \text{relocate}(k_1, \text{f}_{s_2}) :: \text{relocate}(k_2, \text{f}_{k_2} - \text{f}_{s_1} + \text{f}_{s_2}) :: \ldots \\ :: \text{relocate}(k_n, \text{f}_{k_n} - \text{f}_{s_1} + \text{f}_{s_2}) \end{array}}$$

$$\textsf{NS-Exten}\ \frac{s_1 \qquad s_2}{\begin{array}{l} s_1 = s_2 \quad || \\ \text{f}_{s_1} \neq \text{f}_{s_2} \vee \text{l}_{s_1} \neq \text{l}_{s_2} \quad || \\ s_1 = \ldots :: k_1 :: \ldots \wedge s_2 = \ldots :: k_2 :: \ldots \wedge \\ \text{f}_{k_1} = \text{f}_{k_2} \wedge \text{l}_{k_1} = \text{l}_{k_2} \wedge \text{f}_{k_1} \leq \text{l}_{k_1} \wedge k_1 \neq k_2 \ || \\ \text{f}_{s_1} \leq i \leq \text{f}_{s_2} \wedge \text{get}(s_1, i) \neq \text{get}(s_2, i) \end{array}}$$

Fig. 1 Common inference rules for the NS-BASE and NS-EXT calculi

Fig. 2 NS-BASE specific inference rules

$$\textsf{R-Get}\ \frac{v = \text{get}(s, i)}{\begin{array}{l} i < \text{f}_s \vee \text{l}_s < i \quad || \\ \text{f}_s \leq i \leq \text{l}_s \wedge s = k_1 :: \text{const}(i, i, v) :: k_2 \end{array}}$$

$$\textsf{R-Set}\ \frac{s_1 = \text{set}(s_2, i, v)}{\begin{array}{l} (i < \text{f}_{s_2} \vee \text{l}_{s_2} < i) \wedge s_1 = s_2 \quad || \\ \text{f}_{s_1} = \text{f}_{s_2} \wedge \text{l}_{s_1} = \text{l}_{s_2} \wedge \text{f}_{s_2} \leq i \leq \text{l}_{s_2} \wedge \\ s_1 = k_1 :: \text{const}(i, i, v) :: k_3 \wedge s_2 = k_1 :: k_2 :: k_3 \end{array}}$$

5.3 The extended calculus

The extended calculus consists of the rules in Figures 1 and 3. It differs from the base calculus by handling the get and set functions similarly to their treatment in the array decision procedure described in [9]. The Get-Intro rule introduces a get operation from a set operation. The Get-Set rule, commonly referred to as the read-over-write or select-over-store rule in the Array theory, ensures that a get operation applied over a set operation returns the right value.

The Set-Bound rule ensures that a set operation is either performed within the bounds of the target n-sequence or produces an n-sequence equivalent to the original one on which set was applied. The Get-Concat, Set-Concat, and Set-Concat-Inv rules illustrate how the

$$\text{Get-Concat}\ \frac{v = \text{get}(s,i) \qquad s = w_1 :: \ldots :: w_n}{\begin{array}{l} i < \text{f}_s \vee \text{l}_s < i \quad || \\ \text{f}_{w_1} \leq i \leq \text{l}_{w_1} \wedge get(w_1,i) = v \quad || \quad \ldots \quad || \\ \text{f}_{w_n} \leq i \leq \text{l}_{w_n} \wedge get(w_n,i) = v \end{array}}$$

$$\text{Set-Concat}\ \frac{s_1 = \text{set}(s_2,i,v) \qquad s_2 = w_1 :: \ldots :: w_n}{\begin{array}{l} i < \text{f}_{s_2} \vee \text{l}_{s_2} < i \quad || \\ s_1 = \text{set}(w_1,i,v) :: \ldots :: w_n \wedge \text{f}_{w_1} \leq i \leq \text{l}_{w_1} \quad || \quad \ldots \quad || \\ s_1 = w_1 :: \ldots :: \text{set}(w_n,i,v) \wedge \text{f}_{w_n} \leq i \leq \text{l}_{w_n} \end{array}}$$

$$\text{Set-Concat-Inv}\ \frac{s_1 = \text{set}(s_2,i,v) \qquad s_1 = w_1 :: \ldots :: w_n}{\begin{array}{l} i < \text{f}_{s_2} \vee \text{l}_{s_2} < i \quad || \\ s_2 = k :: \ldots :: w_n \wedge \text{f}_{w_1} \leq i \leq \text{l}_{w_1} \wedge w_1 = set(k,i,v) \quad || \quad \ldots \quad || \\ s_2 = w_1 :: \ldots :: k \wedge \text{f}_{w_n} \leq i \leq \text{l}_{w_n} \wedge w_n = set(k,i,v) \end{array}}$$

$$\text{Get-Const}\ \frac{s = \text{const}(f,l,v) \qquad u = \text{get}(s,i)}{i < \text{f}_s \vee \text{l}_s < i \quad || \quad \text{f}_s \leq i \leq \text{l}_s \wedge u = v}$$

$$\text{Get-Set}\ \frac{s_1 = \text{set}(s_2,i,v) \qquad u = \text{get}(s_1,j)}{\begin{array}{l} i < \text{f}_{s_1} \vee \text{l}_{s_1} < i \vee j < \text{f}_{s_1} \vee \text{l}_{s_1} < j \quad || \\ \text{f}_{s_1} \leq i \leq \text{l}_{s_1} \wedge \text{f}_{s_1} \leq j \leq \text{l}_{s_1} \wedge i = j \wedge u = v \quad || \\ \text{f}_{s_1} \leq i \leq \text{l}_{s_1} \wedge \text{f}_{s_1} \leq j \leq \text{l}_{s_1} \wedge i \neq j \wedge u = \text{get}(s_2,j) \end{array}}$$

$$\text{Set-Bound}\ \frac{s_1 = \text{set}(s_2,i,v)}{\begin{array}{l} s_1 = s_2 \quad || \\ \text{f}_{s_1} = \text{f}_{s_2} \wedge \text{l}_{s_1} = \text{l}_{s_2} \wedge \text{f}_{s_1} \leq i \leq \text{l}_{s_1} \wedge \text{get}(s_2,i) \neq v \end{array}}$$

$$\text{Get-Intro}\ \frac{s_1 = \text{set}(s_2,i,v)}{i < \text{f}_{s_1} \vee \text{l}_{s_1} < i \quad || \quad \text{f}_{s_1} \leq i \leq \text{l}_{s_1} \wedge v = \text{get}(s_1,i)}$$

$$\text{Get-Reloc}\ \frac{v = \text{get}(s_1,i) \qquad s_1 s_1 =_{reloc} s_2}{i < f_s \vee l_s < i \quad || \quad f_s \leq i \leq l_s \wedge v = \text{get}(s_2, i - \text{f}_{s_1} + \text{f}_{s_2})}$$

Fig. 3 NS-EXT specific inference rules

get and set operations are handled when applied to an n-sequence in normal form, where the operations affect the right component of the normal form. The Get-Const rule addresses the special case where a get operation is applied to a constant n-sequence. Lastly, the Get-Reloc rule enables the propagation of constraints on elements of an n-sequence to others that are equivalent modulo relocation to it.

6 Cacluli soundness proofs

We place ourselves in a CP (Constraint Programming) context in which each term is associated to a set of domains that are refined throughout the reasoning. The reasoning is done by applying inference rules until saturation, doing constraint propagation as well as decisions. A problem is unsatisfiable if exhaustive exploration of the resolution space leads to no solution, meaning that all orders of decisions and propagations lead to contradictions. Contradictions occur when a term ends up with domains which cannot satisfy the constraints of the problem. A problem is satisfiable if there exist a resolution path in which the domains of the terms allow for the computing of a model that satisfies the constraints of the problem, and it is only when a model is explicitly computed and checked to be satisfiable that a problem can be answered as satisfiable.

In this section, we prove the soundness of the inference rules that constitute the NS-BASE and NS-EXT calculi. We say that a rule is sound if by applying it when its premises are respected, it produces an equisatisfiable environment to the one before its application. That is verified by proving that the consequences of the inference rules can in fact be deducted from their premises.

6.1 Common calculus soundness

In this section we prove the soundness of the rules in Figure 1.

Proof Const-Bounds is sound

Given $s = \mathrm{const}\,(f, l, v)$, the rule just sets the bounds for the n-sequence s to f and l, following the semantics of the const function. □

Proof Reloc-Bounds is sound

Given $s_1 = \mathrm{relocate}\,(s_2, i)$, the rule states:

- if $i = \mathrm{f}_{s_2}$ then $s_1 = s_2$, which is sound by Definition 4 to defined the bounds and Definition 3 to prove equality.
- otherwise it sets the bounds of s_1 and adds the relation $s_1 =_{reloc} s_2$, which is sound by Definition 4. □

Proof Reloc-Inv is sound

Given $s_1 =_{reloc} s_2$, the rule propagates that $s_1 = s_2$ if they have the same first index, and that $s_1 \neq s_2$ otherwise, which are sound by Definitions 4 and 3. □

Proof NS-Slice is sound

Given $s_1 = \text{slice}\,(s, f, l)$ the rule states that if $f < \text{f}_s$ or $l < f$ or $\text{l}_s < l$, then $s_1 = s$, otherwise the rule introduces two n-sequences fresh variables k_1 and k_2 such that $s = k_1{::}s_1{::}k_2$ which amounts to stating that s_1 is equal to the section of the n-sequence s that is within the bounds f and l which are the bounds of s_1, which follows the semantics of the slice function. □

Proof NS-Concat is sound

Given $s = \text{concat}\,(s_1, s_2)$ the rule states that if s_1 is empty then $s = s_2$, if s_2 is empty or $\text{f}_{s_2} \neq \text{l}_{s_1} + 1$ then $s = s_1$, otherwise $s = s_1{::}s_2$ which corresponds to the semantics of the concat function. □

Proof NS-Update is sound

Given $s_1 = \text{update}\,(s_2, s)$ the rule states that if $\text{l}_s < \text{f}_s$ or $\text{f}_s < \text{f}_{s_2}$ or $\text{l}_{s_2} < \text{l}_s$, then $s_1 = s_2$, otherwise it introduces three n-sequences fresh variables k_1, k_2 and k_3 such that $s_1 = k_1{::}s{::}k_3$ and $s_2 = k_1{::}k_2{::}k_3$, stating that s_1 shares the same elements with s_2 on all indices outside the bounds s, wherein s_1 has the same elements as s, while s_2 elements within the bounds of s are those of k, which corresponds to the semantics of the update function. □

Proof NS-Comp-Reloc is sound

Given $s_1 = k_1{::}k_2{::}\ldots{::}k_n$ and $s_1 =_{reloc} s_2$, the rule states that if $\text{f}_{s_1} = \text{f}_{s_2}$ then $s_1 = s_2$, otherwise it states that: $s_2 = relocate(k_1, \text{f}_{s_2}){::}$ $relocate(k_2, \text{f}_{k_2} - \text{f}_{s_1} + \text{f}_{s_2}){::}\ldots{::}relocate(k_n, \text{f}_{k_n} - \text{f}_{s_1} + \text{f}_{s_2})$, which corresponds to computing a normal form for s_2 by relocation that of s_1, which is sound by Definitions 4 and 5. □

Proof NS-Exten is sound.

Given two n-sequences s_1 and s_2, the rule states that they are either equal or unequal, with uneqaulity being presented in three cases, the first case is when the two n-sequences have distinct bounds, the second case is when the normal forms of the two n-sequences contain two distinct components which have the same bounds, and the third case is when there exists an index within the bounds of the two n-sequences in which they hold distinct elements, which is sound by Definition 3. □

6.2 NS-BASE soundness

In this section we prove the soundness of the rules in Figure 2.

Proof R-Get is sound

Given $v = \text{get}(s, i)$, the rule does nothing if i is outside the bounds of s, otherwise it states that $s = k_1 :: \text{const}(i, i, v) :: k_2$ with k_1 and k_2 as fresh n-sequence variables, which amounts to stating that the element at the ith index in s is equal to v, and which corresponds to the semantics of the get function. □

Proof R-Set is sound

Given $s_1 = \text{set}(s_2, i, v)$, the rule states:

- If i is within the bounds of s_2, then the fresh n-sequence variables k_1, k_2 and k_3 are introduced and $s_1 = k_1 :: const(i, i, v) :: k_3 \wedge s_2 = k_1 :: k_2 :: k_3$ is propagated. It states that at the ith index, s_1 contains $\text{const}(i, i, v)$, while s_2 contains k_2. And on the other indices, s_1 and s_2 share the same elements, corresponding to the elements contained in the n-sequence variables k_1 and k_3. That is sound by the semantics of the set function and Definition 5.
- If i is outside the bounds of s_2, then $s_1 = s_2$, which is sound by the semantics of the set function. □

6.3 NS-EXT soundness

In this section we prove the soundness of the rules in Figure 3.

Proof Get-Concat is sound

Given $v = \text{get}(s, i)$ and $s = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s, otherwise it states that $\text{get}(w_m, i) = v$ such that $1 \leq m \leq n$, w_m is one of the components $w_1 :: \ldots :: w_n$ and $\text{f}_{w_m} \leq i \leq \text{l}_{w_m}$. It amounts to stating that the ith element of s is equal to the ith element of the component of s's n-sequence normal form that encompasses the i index, which is sound by the semantics of the get function Definition 5. □

Proof Set-Concat is sound

Given $s_1 = \text{set}(s_2, i, v)$ and $s_2 = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s_2, otherwise it sets the n-sequence normal form of s_1 to the same as s_2 except on the component w_m, such that w_m is within $w_1 :: \ldots :: w_n$ and $\text{f}_{w_m} \leq i \leq \text{l}_{w_m}$, which is replaced by $\text{set}(w_m, i, v)$, which amounts to applying the set function to the component of s_2's normal that encompasses the index i, which is sound by the definition of set and Definition 5. □

Proof Set-Concat is sound

Given $s_1 = \text{set}(s_2, i, v)$ and $s_1 = w_1 :: \ldots :: w_n$, the rule does nothing if i is outside the bounds of s_2, otherwise it sets the n-sequence normal form of s_2 to the same as s_1 except on the component w_m, such that w_m is within $w_1 :: \ldots :: w_n$ and $\text{f}_{w_m} \leq i \leq \text{l}_{w_m}$, which is replaced by a fresh n-sequence variable k, such that $w_m = \text{set}(k, i, v)$, which amounts to saying that s_2 has the same normal form components as s_1 except on the component with encompasses the index i, which is sound by the definition of set and Definition 5. □

Proof Get-Const is sound

Given $s = \text{const}(f, l, v)$ and $u = \text{get}(s, i)$, if i is within the bounds of s, then $u = v$ since s is a constant n-sequence, otherwise the rule does nothing, which is sound by the semantics of the get and const functions. □

Proof Set-Bound is sound

Given $s_1 = \text{set}(s_2, i, v)$, the rule states that:

- Either $s_1 = s_2$, due to i being outside the bounds of s_2 or because $v = \text{get}(s_2, i)$, which is sound by the semantics of the set and get functions
- Or i is within the bounds of s_2, s_1 and s_2 have equal bounds and $v \neq \text{get}(s_2, i)$, which is sound by the semantics of the set and get functions □

Proof Get-Set is sound

Given $s_1 = \text{set}(s_2, i, v)$ and $u = \text{get}(s_1, j)$, if i is not within the bounds of s_1 then the rule does nothing, otherwise:

- If i is within the bounds of s_2 and $i = j$ then $u = v$, which is sound by the semantics of the get and set functions.
- If i is within the bounds of s_2 and $i \neq j$ then $u = \text{get}(s_2, j)$, which is sound by the semantics of the get and set functions. □

Proof Get-Intro is sound

Given $s_1 = \text{set}(s_2, i, v)$, the rule states that if i is within the bounds of s_2, then $v = \text{get}(s_1, i)$, otherwise the rule does nothing, which is sound by the semantics of the functions get and set. □

Proof Get-Reloc is sound

Given $v = get(s_1, i)$ and $s_1 =_{reloc} s_2$, the rule does nothing if i is not within the bounds of s_1, otherwise it states that $v = \text{get}(s_2, i - \text{f}_{s_1} + \text{f}_{s_2})$ which is sound by Definition 4. □

7 Implementation

To evaluate the performance of the calculi described in the previous section, we implemented them in Colibri2. Colibri2 is a CP (Constraint Programming) solver used to reason about SMT formulas across various theories, including linear and non-linear integer and real arithmetic, floating-point arithmetic, fixed-size bit-vectors, and arrays. Colibri2 is a reimplementation and extension of the COLIBRI [15] CP solver. Unlike SAT and SMT solvers, Colibri2 does not support clause learning. However, it provides greater control over the scheduling of propagations

for theory developers and simplifies theory combination. Before declaring a formula satisfiable, Colibri2 explicitly computes a model and verifies it against the input formula, avoiding the need for a combination framework to maintain model soundness. Nevertheless, a theory combination framework can still be useful to make computing satisfiable models more efficient.

In Colibri2, a term from any theory can be associated with an arbitrary number of domains, each domain holding some specific information about the term. For example, in the case of arithmetic terms, the interval union domain is used represent all possible values the term can take. When the equivalence classes of two terms are merged, their domains are also merged. Any theory in Colibri2 can perform constraint propagation, which involves pruning the domains of terms and propagating updates to other parts of the system. This ensures that other theories using the terms affected by the constraint are informed of the updated domain and can act on this information. Constraint propagation can take the form of assigning a value to a term or refining one of its domains. For instance, in integer arithmetic, propagating the constraint $x \geq 0$ for an integer term x restricts its interval domain to $[0, +\infty)$.

Colibri2 also supports semantic decisions. A decision involves registering a backtracking point and branching based on the assumption that a given proposition holds. In a given problem, the proposition is assumed to hold, and satisfiability is checked under this assumption. If the problem is satisfiable, it is solved. Otherwise, the solver backtracks to the backtracking point and explores a new branch where an alternative assumption, typically the negation of the first one, holds. Decisions can also involve constraints on terms, encompassing multiple complementary constraints over one or more terms.

When neither propagations nor decisions can be made, another phase, called the last effort phase, is started. The last effort phase is another propagation phase that is used to do costlier propagations that we don't want to do in the first propagation phase. These include calls to the simplex algorithm and quantifier instantiations that create new terms. The last effort phase can also introduce new decisions, or even additional propagations for the next last effort phases.

The implementation of the scheduler uses an efficient time-wheel data structure (similar to §6.2 in [18]), which allows for a good trade-off between prioritizing higher-priority propagations and ensuring fairness. Prioritizing faster propagations is often beneficial, but such propagations can sometimes loop and lead to slow convergence. For example, local interval propagation starting with $x, y \in [0, 2^{32}]$ with constraints $x < y$ and $y < x$ will remove only one integer from the domain per propagation. Using the time-wheel data structure ensures low-priority propagations are scheduled and executed before the slow convergence completes.

The scheduler also ensures that last effort propagations that are found to be useful, as they lead to contradictions, are promoted to the main propagation phase after backtracking. These propagations are then executed earlier instead of staying in the last effort phase.

In this section, we discuss our implementation choices, particularly how equivalence modulo relocation (cf. Definition 4), is represented and used, as well as how the inference rules that formalize the calculi are applied in practice.

7.1 Equivalence modulo relocation

As described in Section 5.1, the $=_{reloc}$ relation links pairs of n-sequences that contain the same sequences of elements but have different starting indices. Proposition 1 asserts that the $=_{reloc}$ relation is an equivalence relation. Therefore, we say that n-sequence terms that are

equivalent modulo relocation to one another, eventually transitively, are in the same $=_{reloc}$ class. We present two ways to handle these classes of n-sequences.

7.1.1 Equivalence modulo relocation with undirected graphs

A straightforward way to represent such relations is to use an undirected graph in which the vertices represent n-sequences, and an edge between two vertices indicates that the two n -sequences are equivalent modulo relocation. The graph is undirected because the $=_{reloc}$ relation is an equivalence relation (cf. Proposition 1).

Equivalence modulo relocation is used for constraint propagation, whether of n-sequence normal forms through the NS-Comp-Reloc rule in Figure 1 or of constraints on elements through the Get-Reloc rule in Figure 3. It is also used for equality detection through the Reloc-Inv rule in Figure 1. To efficiently perform constraint propagation over the $=_{reloc}$ relation, it is necessary to retrieve the $=_{reloc}$ class of any given n-sequence.

Using a graph data structure, constraint propagation can be achieved through graph exploration algorithms, such as Breadth-First Search, to find all elements of the $=_{reloc}$ class of a given n-sequence, or by using an auxiliary data structure that associates to every n-sequenceall the elements of its $=_{reloc}$ class. However, these approaches are costly, the first due to the time complexity of graph exploration, and the second due to the memory and time required for maintaining the auxiliary data structure.

7.1.2 Equivalence modulo relocation with a labeled union-find data structure

We opted for a different approach based on the labeled union-find data structure [13]. This structure extends the traditional union-find by labeling the edges in the trees that represent equivalence classes. It is used to represent equivalence relations parameterized by the aforementioned labels. These labels must satisfy the group axioms: they must be composable, have a neutral element, and have an inverse.

In our case, the set of nodes $\mathbb{N}$ in the labeled union-find data structure corresponds to the set of n-sequence terms, and the set of labels $\mathbb{L}$ corresponds to the set of integer polynomials that represent the differences in starting indices between n-sequence terms. Integer polynomials satisfy the group axioms: composition is integer addition ($\mathrm{comp}(x, y) = x + y$), the neutral element is 0 ($\forall x.\ x + 0 = 0 + x = x$), and the inverse function is negation ($\mathrm{inv}(x) = -x$). Since the labels respect the group axioms, we ensure that all paths are compressed in our labeled union-find data structure.

Example 1 Given the formulas $F_1 : s_1 = \mathrm{relocate}(s, k_1)$, $F_2 : s_2 = \mathrm{relocate}(s, k_2)$, and $F_3 : s_3 = \mathrm{relocate}(s_2, k_3)$, and assuming that s is chosen as the representative:

- F_1: An edge labeled with $\mathrm{f}_s - \mathrm{f}_{s_1}$ is added from s_1 to s.
- F_2: An edge labeled with $\mathrm{f}_s - \mathrm{f}_{s_2}$ is added from s_2 to s.
- F_3: Given that the distance difference between s_3 and s_2 is $\mathrm{f}_{s_2} - \mathrm{f}_{s_3}$, and that the label on the edge from s_2 to s is $\mathrm{f}_s - \mathrm{f}_{s_2}$, then an edge labeled with $\mathrm{f}_s - \mathrm{f}_{s_3}$ is added from s_3 to s.

Concretely, the implementation is as follows: each n-sequence is either a representative or a non-representative of a $=_{reloc}$ class. Each representative s is associated to a map $\{k_1 \mapsto s_1, k_2 \mapsto s_2, \ldots\}$, where $s_1, s_2, \ldots$ are non-representative n-sequences in the $=_{reloc}$ class of s, and $k_1, k_2, \ldots$ are the labels on the edges from $s_1, s_2, \ldots$ to s, respectively. Each non-representative s_i is associated with a pair (s, k_i), where s is the representative of its $=_{reloc}$ class and k_i is the label of the edge from s_i to s.

The representative of a $=_{reloc}$ class is initially chosen arbitrarily and remains unchanged when adding new elements to the $=_{reloc}$ class, unless an element from another class is added, in which case the representative of the larger class (i.e., the one with more elements) is chosen as the representative of the merged class.

In addition to path compression, we maintain normalized edge labels. This is possible because Colibri2 associates a normalized polynomial with each arithmetic term. This makes equality detection straightforward. For example, given a representative s associated with a map $\{k_1 \mapsto s_1, k_2 \mapsto s_2\}$, if a new non-representative s_3 is added with the label k_3 such that $k_1 = k_3$, then we can deduce $s_1 = s_3$. Furthermore, for each representative s, we add $0 \mapsto s$ to its map of non-representatives, ensuring that if a new term s_0 is added with the label 0, we can directly deduce $s_0 = s$.

This implementation also simplifies constraint propagation: retrieving all members of a class is as simple as accessing the map of the class's representative. If a constraint is propagated to a representative, the map of non-representatives can be accessed directly. If it is propagated to a non-representative, the constraint must first be applied to the representative, then propagated from the representative to the other non-representatives.

To improve efficiency, we restrict constraint propagation to occur only from non-representatives to representatives. This ensures that all constraints on a given n-sequence are always propagated to the representative of its $=_{reloc}$ class. When two classes are merged, and a new representative is chosen, the constraints from the other (now former) representative are also propagated to the new one. This approach effectively computes the reduced product of the constraints for all members of a $=_{reloc}$ class by only propagating to the representative, instead of redundantly propagating constraints to every member of the class.

7.2 Simplification rewrites

To ensure that Assumption 1 is maintained, we implemented a callback system. Callbacks are functions that are executed whenever a specific event occurs. In the case of Assumption 1, the callbacks correspond to simplifications, and the events defined for a given n-sequence s are one of the following:

- s is empty: this event occurs when the proposition $\mathrm{l}_s < \mathrm{f}_s$ is determined to be true.
- s is not flat: this event occurs when s is determined to be equal to any normal form of the shape _::_.

Whenever one of these two events occurs, the corresponding simplification is applied. When a simplification is applied to a given n-sequence s, it is also applied to all the n-sequences that are in its $=_{reloc}$ class. For instance, when an n-sequence s is determined to be empty, it is removed from all the n-sequence normal forms in which it occurs. Similarly, all elements

of its $=_{reloc}$ class are also removed from all the n-sequence normal forms in which they appear. This ensures that Assumption 1 is consistently upheld.

7.3 Reasoning

Most of the inference rules of the reasoning are applied as soon as possible. When a rule is applied, it is necessary to first determine if a decision needs to be made to know which of the inference rule's consequences should be applied. For instance, when the term $s_1 = \text{relocate}(s_2, i)$ is encountered, if it is known that $i = \text{f}_{s_2}$ is true, then $s_1 = s_2$ is propagated immediately. Otherwise, a decision is registered on whether $i = \text{f}_{s_2}$ is true to determine which of the rule's consequences should be propagated.

This applies to the rules Const-Bounds, Reloc-Bounds, NS-Slice, Reloc-Inv, NS-Concat, NS-Update, NS-Split, and NS-Comp-Reloc in Figure 1, as well as all the rules in Figures 2 and 3.

In contrast, the NS-Exten rule is applied only during the last effort phase. This means that it is applied to all pairs of n-sequences that are not known to be equal, but only after all other rules, their propagations and decisions, have been executed. When the NS-Exten rule is applied, it can introduce new propagations and decisions, which in turn may introduce further ones, and so forth. In such cases, the NS-Exten rule is reapplied only after all newly introduced propagations and decisions are completed. If no new propagations or decisions are introduced, the NS-Exten rule is not reapplied.

7.4 Support for the Seq theory

To ensure compatibility with the Seq theory of cvc5 and Z3, we added support for a subset of their versions of the Seq theory. This subset consists of the common sequence operations that are used to represent array-like data structures in programming languages. These operations include: seq.unit, seq.len, seq.nth, seq.update, seq.extract, and seq.++.

To support these operations in our solver, we simply internally translate them into operations from the NSeq theory as follows:

- Sequence terms: n-sequence terms where the first index is 0 and the last index is greater than or equal to -1.
- seq.empty: We added the nseq.empty symbol, representing a constant empty NSeq term where the first index is 0 and the last index is -1.
- $\text{seq.unit}(v)$: $\text{const}(0, 0, v)$
- $\text{seq.len}(s)$: $\text{l}_s - \text{f}_s + 1$
- $\text{seq.nth}(s, i)$: $\text{get}(s, i)$
- $\text{seq.update}(s_1, i, s_2)$:

$$\begin{array}{c}\text{let}\,(r,\,\text{relocate}\,(s_2, i),\,\text{ite}\,(\text{f}_{s_1} \le i \le \text{l}_{s_1} \wedge \text{l}_{s_1} < \text{l}_r,\\ \text{update}\,(s_1,\,\text{slice}\,(r, i,\,\text{l}_{s_1})),\,\text{update}\,(s_1, r)))\end{array}$$

- $\text{seq.extract}(s, i, j)$:

$$\text{ite}\,(i < \text{f}_s \vee \text{l}_s < i \vee j \le 0, \epsilon,\,\text{slice}\,(s, i, \min(\text{l}_s, i + j - 1)))$$

- $\mathrm{seq}.{+}{+}(s_1, s_2, s_3, \ldots, s_n)$:

$$\begin{array}{l}\mathrm{let}\,(c_1,\,\mathrm{concat}\,(s_1,\,\mathrm{relocate}\,(s_2,\,1_{s_1}+1)),\\ \mathrm{let}\,(c_2,\,\mathrm{concat}\,(c_1,\,\mathrm{relocate}\,(s_3,\,1_{c_1}+1)),\\ \qquad\qquad\ldots\\ \mathrm{concat}\,(c_{n-2},\,\mathrm{relocate}\,(s_n,\,1_{c_{n-2}}+1))))\end{array}$$

8 Experimental evaluation

In this section, we present experimental results of our implementations described in Section 7, of the calculi described in Section 5. These experiments were conducted on quantifier-free benchmarks that use only the Seq and NSeq theories with the theory of uninterpreted functions. The benchmarks are a subset of those used by Sheng et al. [17], which were originally translated into the Seq theory from the QF_AX SMT-LIB benchmarks [4]. We also translated the QF_AX benchmarks into the NSeq theory to test our native calculi and compare them with the encoding of the NSeq theory using the Seq and ADT theories described in Section 4.

Our implementations in Colibri2[1] of the NS-BASE and NS-EXT calculi can be used with the following commands:

- NS-BASE: colibri2 --nseq-base
- NS-EXT: colibri2 --nseq-ext

For comparison, we used cvc5 (version 1.2.0) and Z3 (version 4.13.3) as reference solvers. We tested three configurations of cvc5, each using a different strategy for handling sequence operations:

- cvc5: cvc5
- cvc5-eager: cvc5 --seq-arrays=eager
- cvc5-lazy: cvc5 --seq-arrays=lazy
- z3: z3

Figures 4 and 5 illustrate the number of goals solved over accumulated time for Seq and NSeq benchmarks, respectively. Detailed statistics including number of goals solved, timeouts, errors, and runtime metrics (average, median, and total solving time) are shown in Tables 2 to 5.

8.1 Translated n-sequence benchmarks

Similarly to what was done in [17], we translated QF_AX benchmarks into the NSeq theory by replacing the sort of indices with integers, the sort of arrays with the sort of n-sequences, and the array operations select and store with the n-sequence operations get and set, respectively.

To compare the native calculi approach with the encoding approach described in Section 4, we also created a version of the benchmarks in which, in addition to translating the

[1] Available at: https://git.frama-c.com/pub/colibrics/-/tree/acta_informatica_2024 (commit SHA: 8d654690eb5c08643a87f0e41334f66311186e40)

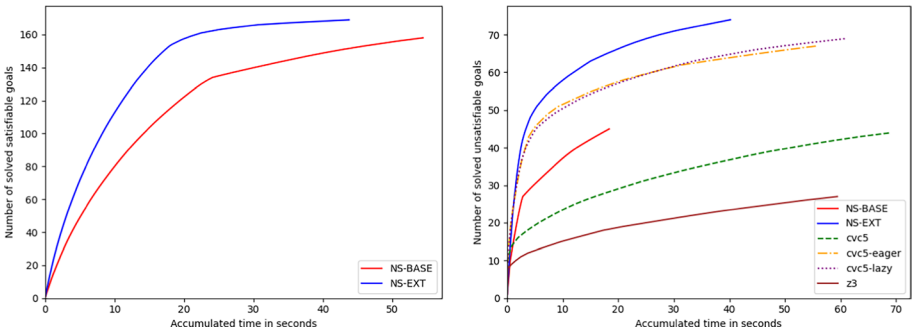

Fig. 4 Number of solved goals by accumulated time in seconds on quantifier-free NSeq benchmarks translated from the QF_AX SMT-LIB benchmarks

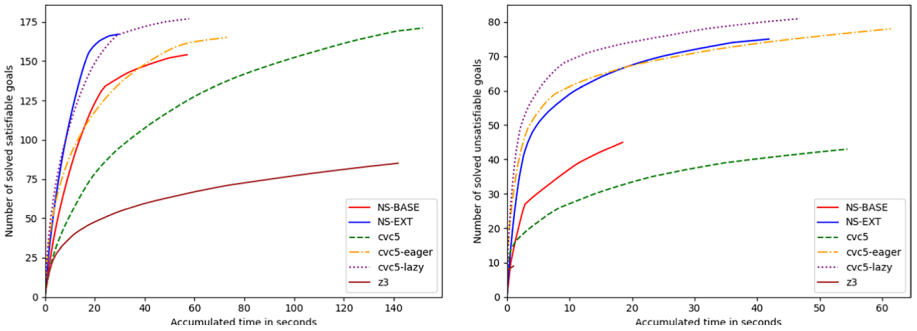

Fig. 5 Number of solved goals by accumulated time in seconds on quantifier-free Seq benchmarks translated from the QF_AX SMT-LIB benchmarks

Table 2 Statistics on the performance of the solvers on quantifier-free unsatisfiable NSeq benchmarks

Solver	Solved	Timeout	Error	Unknown	Avg. Time	Med. Time	Tot. Time
NS-BASE	45	115	4	0	0.408	0.138	18.346
NS-EXT	74	90	0	0	0.543	0.102	40.159
cvc5	44	120	0	0	1.570	1.279	69.064
cvc5-eager	67	97	0	0	0.832	0.139	55.711
cvc5-lazy	69	95	0	0	0.886	0.156	61.156
z3	22	26	0	0	1.949	1.399	42.889

benchmarks, we added the definitions of the NSeq theory operations using the operations of the Seq and ADT theories.

Figure 4 shows that NS-EXT performs better overall on unsatisfiable benchmarks, solving more goals faster than all other solvers. While NS-BASE performs better than both cvc5 and z3, though it trails behind NS-EXT, cvc5-eager and cvc5-lazy.

Table 2 confirms that NS-EXT achieves the best overall result (74 goals solved) with one of the lowest average runtimes (0.543s). Notably, NS-BASE solves slightly more problems than cvc5 (45 vs. 44) with a significantly lower average runtime (0.138 vs. 1.570).

Table 3 Statistics on the performance of the solvers on quantifier-free satisfiable NSeq benchmarks

Solver	Solved	Timeout	Error	Unknown	Avg. Time	Med. Time	Tot. Time
NS-BASE	158	229	0	0	0.344	0.184	54.382
NS-EXT	169	218	0	0	0.259	0.113	43.753

Since these benchmarks originate from array benchmarks and contain many get and set operations which, as shown in the rules in Figure 2, require multiple decisions and introduce n -sequence normal forms with small n-sequence components, the solver requires significant computation time and does not scale very well on these benchmarks. Reasoning over such problems would work better with clause learning which would allow a better handling of decisions.

Regarding satisfiable goals, only Colibri2 managed to determine Satisfiability within the time limit, with NS-EXT clearly surpassing NS-BASE in both speed and the number of goals solved. Table 3 shows that the average runtime of NS-EXT is also better than the one of NS-BASE.

8.2 Translated sequence benchmarks

As mentioned in Section 7.4, we implemented support for the Seq theory by encoding it on top of the NSeq theory. To evaluate the performance of our support for the Seq theory, we compared it with the Seq theories of cvc5 and Z3 on Seq benchmarks, which were translated from QF_AX benchmarks.

The graph on the right in Figure 5 shows that on unsatisfiable goals, our NS-EXT implementation outperforms cvc5 and z3 in both time and the number of goals solved. Meanwhile, NS-BASE initially solves more goals than cvc5, but solves fewer overall. Additionally, cvc5-eager and cvc5-lazy perform better than the other solvers.

As seen in Table 4, cvc5-lazy solves the most goals and achieves a relatively low average runtime. However, NS-EXT solves nearly as many while maintaining a slightly lower average runtime.

In the satisfiable case, Table 5 shows that cvc5-lazy solves the most goals (177), followed by cvc5-eager (171), while NS-EXT solves slightly fewer (167) but with a much lower average time per goal (0.176s vs. 0.327s and 0.443s). These trends are reflected in Figure 5, where the NS-EXT curve is steeper early on but levels off before reaching the maximum. A similar trend can be noticed with NS-BASE compared to cvc5-eager, the two curves are close to one another and cross each other at two points, before cvc5-eager takes over.

8.3 Discussion

In the context of program verification, performance on unsatisfiable goals is of greater importance, although the satisfiable cases remain valuable. Since Colibri2 constructs concrete models before concluding satisfiability, we aim to improve our current model generation technique for n-sequences.

For unsatisfiable goals, our solver performs competitively with state-of-the-art SMT solvers such as cvc5 and Z3. However, we have observed that certain goals which remain unsolved within a short timeout (e.g., 5 seconds) also remain unsolved even with significantly longer timeouts. This suggests potential performance bottlenecks in our propagators for the NSeq theory.

Table 4 Statistics on the performance of the solvers on quantifier-free unsatisfiable Seq benchmarks

Solver	Solved	Timeout	Error	Unknown	Avg. Time	Med. Time	Tot. Time
NS-BASE	45	117	2	0	0.410	0.139	18.449
NS-EXT	75	89	0	0	0.558	0.107	41.830
cvc5	43	121	0	0	1.265	0.733	54.392
cvc5-eager	78	86	0	0	0.787	0.108	61.367
cvc5-lazy	81	83	0	0	0.578	0.064	46.806
z3	9	39	0	0	0.112	0.018	1.007

Table 5 Statistics on the performance of the solvers on quantifier-free satisfiable Seq benchmarks

Solver	Solved	Timeout	Error	Unknown	Avg. Time	Med. Time	Tot. Time
NS-BASE	154	232	1	0	0.370	0.175	56.976
NS-EXT	167	220	0	0	0.176	0.108	29.363
cvc5	171	216	0	0	0.887	0.603	151.742
cvc5-eager	165	222	0	0	0.443	0.219	73.080
cvc5-lazy	177	210	0	0	0.327	0.150	57.821
z3	85	418	0	0	1.668	1.014	141.773

A notable pattern visible in Figures 4 and 5 is the presence of inflection points in the performance curves of NS-BASE and NS-EXT. These may indicate that the solver struggles with specific classes of problems, warranting further investigation.

It is also worth noting that our translation from Seq to NSeq in Colibri2 often introduces more complex terms. Additionally, Colibri2 does not currently implement clause learning, which can make the search space exploration more costly compared to other SMT solvers.

9 Conclusion

In this paper, we explored the topic of reasoning over n-indexed sequences in SMT. We proposed a theory for such sequences and discussed approaches for reasoning over it, whether by using existing theories or by adapting calculi from the theory of sequences to this theory and implementing the calculi in a solver. We discussed the various changes we had to bring to the calculi to adapt them to our theory and mentioned different implementation details that helped us obtain competitive performance results with state-of-the-art SMT solvers, despite the absence of clause learning.

Looking ahead, we plan on delving deeper into different reasoning approaches for this theory, exploring their respective strengths and weaknesses through benchmarking with n -indexed sequences. We also aim to identify additional use cases, other that representing Ada arrays, where n-indexed sequences appear as a natural choice.

Appendix A. Representation of *n*-indexed sequences using sequences and algebraic data types

```
(declare-datatypes ((NSeq 1))
  ((par (T) ((nseq.mk (nseq.first Int) (nseq.seq (Seq T)))))))

(define-fun nseq.last (par (T) ((s (NSeq T))) Int
  (+ (- (seq.len (nseq.seq s)) 1) (nseq.first s))))

(define-fun nseq.get (par (T) ((s (NSeq T)) (i Int)) T
  (seq.nth (nseq.seq s) (- i (nseq.first s)))))

(define-fun nseq.set
  (par (T) ((s (NSeq T)) (i Int) (v T)) (NSeq T)
    (nseq.mk (nseq.first s)
      (seq.update
        (nseq.seq s) (- i (nseq.first s)) (seq.unit v)))))

(declare-fun nseq.const (par (T) (Int Int T) (NSeq T)))

;; "nseq_const"
(assert (par (T) (forall ((f Int) (l Int) (v T))
    (!
      (let ((s (nseq.const f l v)))
        (and
          (= (nseq.first s) f)
          (= (nseq.last s) l)
          (forall ((i Int))
            (=> (and (<= f i) (<= i l)) (= (nseq.get s i) v)))))
      :pattern ((nseq.const f l v))))))

(define-fun nseq.relocate
  (par (T) ((s (NSeq T)) (f Int)) (NSeq T)
     (nseq.mk f (nseq.seq s))))

(define-fun nseq.concat
  (par (T) ((s1 (NSeq T)) (s2 (NSeq T))) (NSeq T)
    (ite (< (nseq.last s1) (nseq.first s1))
      s2
      (ite
        (or
          (< (nseq.last s2) (nseq.first s2))
          (not (= (nseq.first s2) (+ (nseq.last s1) 1))))
        s1
        (nseq.mk
          (nseq.first s1)
          (seq.++ (nseq.seq s1) (nseq.seq s2)))))))

(define-fun nseq.slice
  (par (T) ((s (NSeq T)) (f Int) (l Int)) (NSeq T)
    (ite
      (and
        (<= f l)
        (and (<= (nseq.first s) f) (<= l (nseq.last s))))
      (nseq.mk f (seq.extract (nseq.seq s) (- f (nseq.first s))
         (+ (- l f) 1)))
      s)))

(define-fun nseq.update
  (par (T) ((s1 (NSeq T)) (s2 (NSeq T))) (NSeq T)
    (ite
      (and
        (<= (nseq.first s2) (nseq.last s2))
        (<= (nseq.first s1) (nseq.first s2))
        (<= (nseq.last s2) (nseq.last s1)))
      (nseq.mk (nseq.first s1)
        (seq.update
          (nseq.seq s1)
          (- (nseq.first s2) (nseq.first s1))
          (nseq.seq s2)))
        s1)))
```

Appendix B. Proof that equivalence modulo relocation is an equivalence relation

In this section we prove the claim that the equivalence modulo relocation relation in an equivalence relation

Proof Proposition 1: The $=_{reloc}$ relation is an equivalence relation.

- (a) Reflexivity: Given an n-sequence s, $s =_{reloc} s$ holds since an n-sequence is equal to itself, therefore equivalent (modulo relocation) to itself by definition.
- (b) Symmetry:

  Given two n-sequences s_1 and s_2, $s_1 =_{reloc} s_2$ implies: $\mathrm{l}_{s_2} = \mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ (1) and
  $\forall i : Int, \mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1} \Rightarrow get(s_1, i) = get(s_2, i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2})$ (2)
  From rearranging (1) we get: $\mathrm{l}_{s_1} = \mathrm{l}_{s_2} - \mathrm{f}_{s_2} + \mathrm{f}_{s_1}$ (3)
  By subtracting $\mathrm{f}_{s_1} - \mathrm{f}_{s_2}$ from the terms of the disequality in (2) we get:
  $\forall i : Int, \mathrm{f}_{s_2} \leq i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2} \leq \mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2} \Rightarrow get(s_1, i) = get(s_2, i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2})$ (4)
  From (2), we can replace $\mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ with l_{s_2} in (4), to get:
  $\forall i : Int, \mathrm{f}_{s_2} \leq i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2} \leq \mathrm{l}_{s_2} \Rightarrow get(s_1, i) = get(s_2, i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2})$ (5)
  if we introduce a variable j such that $j = i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ in (5) we get:
  $\forall j : Int, \mathrm{f}_{s_2} \leq j \leq \mathrm{l}_{s_2} \Rightarrow get(s_1, j - \mathrm{f}_{s_2} + \mathrm{f}_{s_1}) = get(s_2, j)$ (6)
  From (3) and (6), we deduce that $s_2 =_{reloc} s_1$.

- (c) Transitivity:

  Given three n-sequences s_1, s_2 and s_3, $s_1 =_{reloc} s_2$ and $s_2 =_{reloc} s_3$ imply that:
  $\mathrm{l}_{s_2} = \mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ (1)
  $\forall i : Int, \mathrm{f}_{s_1} \leq i \leq \mathrm{l}_{s_1} \Rightarrow get(s_1, i) = get(s_2, i - \mathrm{f}_{s_1} + \mathrm{f}_{s_2})$ (2)
  $\mathrm{l}_{s_3} = \mathrm{l}_{s_2} - \mathrm{f}_{s_2} + \mathrm{f}_{s_3}$ (3)
  $\forall i : Int, \mathrm{f}_{s_2} \leq i \leq \mathrm{l}_{s_2} \Rightarrow get(s_2, i) = get(s_3, i - \mathrm{f}_{s_2} + \mathrm{f}_{s_3})$ (4)
  By replacing l_{s_2} with $\mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ in (3) we get:
  $\mathrm{l}_{s_3} = \mathrm{l}_{s_1} - \mathrm{f}_{s_1} + \mathrm{f}_{s_3}$ (5)
  From (1) we get:
  $\mathrm{l}_{s_1} = \mathrm{l}_{s_2} + \mathrm{f}_{s_1} - \mathrm{f}_{s_2}$ (6)
  By adding $\mathrm{f}_{s_1} - \mathrm{f}_{s_2}$ to the terms of the disequality in (4), we get:
  $\forall i : Int, \mathrm{f}_{s_1} \leq i + \mathrm{f}_{s_1} - \mathrm{f}_{s_2} \leq \mathrm{l}_{s_2} + \mathrm{f}_{s_1} - \mathrm{f}_{s_2} \Rightarrow get(s_2, i) = get(s_3, i - \mathrm{f}_{s_2} + \mathrm{f}_{s_3})$ (7)
  From (6), we can replace $\mathrm{l}_{s_2} + \mathrm{f}_{s_1} - \mathrm{f}_{s_2}$ with l_{s_1} in (7) and get:
  $\forall i : Int, \mathrm{f}_{s_1} \leq i + \mathrm{f}_{s_1} - \mathrm{f}_{s_2} \leq \mathrm{l}_{s_1} \Rightarrow get(s_2, i) = get(s_3, i - \mathrm{f}_{s_2} + \mathrm{f}_{s_3})$ (8)
  By introduction a variable j, such that $i = j - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ and replacing i with $j - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}$ in (8) we get:
  $\forall j : Int, \mathrm{f}_{s_1} \leq j \leq \mathrm{l}_{s_1} \Rightarrow get(s_2, j - \mathrm{f}_{s_1} + \mathrm{f}_{s_2}) = get(s_3, j - \mathrm{f}_{s_1} + \mathrm{f}_{s_3})$ (9)
  From (2) we get:
  $\forall j : Int, \mathrm{f}_{s_1} \leq j \leq \mathrm{l}_{s_1} \Rightarrow get(s_1, j) = get(s_3, j - \mathrm{f}_{s_1} + \mathrm{f}_{s_3})$ (10)

From (5) and (10), we deduce that $s_1 =_{reloc} s_3$.From (a), (b) and (c), we deduce that $=_{reloc}$ is a reflexive, symmetric and transitive relation, it is therefore an equivalence relation. □

Author contributions Hichem Rami Ait El Hara wrote the manuscript with support from François Bobot and Guillaume Bury. Hichem Rami Ait El Hara conceived the original idea. Hichem Rami Ait El Hara and François Bobot participated in the implementation of the tool. Hichem Rami Ait El Hara carried out the experiments with inputs from François Bobot and Guillaume Bury. François Bobot and Guillaume Bury supervised the project.

Funding Open access funding provided by Université Paris-Saclay.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Ait-El-Hara, H.R., Bobot, F., Bury, G.: An SMT Theory for n-Indexed Sequences. In: Reger, G., Zohar, Y. (eds.) Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories. CEUR Workshop Proceedings, vol. 3725, pp. 64–74. CEUR, Montreal, Canada (2024). https://ceur-ws.org/Vol-3725/#short13
2. Ait-El-Hara, H.R., Bobot, F., Bury, G.: On SMT Theory Design: The Case of Sequences. In: Kalpa Publications in Computing, vol. 18, pp. 14–29. EasyChair, Balaclava, Mauritius (2024). https://doi.org/10.29007/75tl. ISSN: 2515-1762
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer, Munich, Germany (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
5. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. FMCAD '17, pp. 55–59. FMCAD Inc, Austin, Texas (2017)
6. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB Format for Sequences and Regular Expressions. Strings (2012)
7. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: CDSAT for Nondisjoint Theories with Shared Predicates: Arrays With Abstract Length. Satisfiability Modulo Theories workshop, CEUR Workshop Proceedings **3185** (2022). Accessed 2024-02-23

8. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 427–442. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11609773_28
9. Christ, J., Hoenicke, J.: Weakly Equivalent Arrays. In: Lutz, C., Ranise, S. (eds.) Frontiers of Combining Systems. Lecture Notes in Computer Science, pp. 119–134. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_8
10. Furia, C.A.: What's Decidable about Sequences? In: Bouajjani, A., Chin, W.-N. (eds.) Automated Technology for Verification and Analysis, pp. 128–142. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_11
11. Ghilardi, S., Gianola, A., Kapur, D., Naso, C.: Interpolation Results for Arrays with Length and MaxDiff. ACM Trans. Comput. Log. **24**(4), 28–12833 (2023). https://doi.org/10.1145/3587161
12. Jeż, A., Lin, A.W., Markgraf, O., Rümmer, P.: Decision Procedures for Sequence Theories. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, pp. 18–40. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_2
13. Lesbre, D., Lemerre, M., Ait-El-Hara, H.R., Bobot, F.: Relational abstractions based on labeled union-find. Proc. ACM Program. Lang. **9**(PLDI) (2025). https://doi.org/10.1145/3729298
14. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
15. Marre, B., Bobot, F., Chihani, Z.: Real Behavior of Floating Point Numbers. In: The SMT Workshop, Heidelberg, Germany (2017). SMT 2017, 15th International Workshop on Satisfiability Modulo Theories. https://cea.hal.science/cea-01795760
16. Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Sheng, Y., Nötzli, A., Reynolds, A., Zohar, Y., Dill, D., Grieskamp, W., Park, J., Qadeer, S., Barrett, C., Tinelli, C.: Reasoning About Vectors: Satisfiability Modulo a Theory of Sequences. J. Autom. Reason. **67**(3), 32 (2023). https://doi.org/10.1007/s10817-023-09682-2
18. Varghese, G., Lauck, T.: Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, vol. 21, pp. 25–38. Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/37499.37504
19. Wang, Q., Appel, A.W.: A Solver for Arrays with Concatenation. J. Autom. Reason. **67**(1), 4 (2023). https://doi.org/10.1007/s10817-022-09654-y