An SMT Theory for n-Indexed Sequences

Hichem Rami Ait El Hara[1,2], François Bobot[2] and Guillaume Bury[1]

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

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 works suggest that developing tailored reasoning for some theories, such as sequences and strings, is 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 the 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. Sequences are dynamically sized, and their theory has a rich signature. It allows selecting elements of a sequence by their index, concatenating two sequences, extracting sub-sequences, and performing other operations. The expressiveness of the theory of sequences makes it easier to represent various commonly found data structures in programming languages, such as arrays in the C language, lists in Python, etc.

The theory of arrays is less expressive as it only supports selecting and storing one value at one index at a time, and arrays have fixed sizes determined by the number of inhabitants of the sort of indices. In contrast, sequences have dynamic lengths and operations allowing the selection and updating of sets of indices at a time. To use the theory of arrays to represent sequences, one would need to extend it and axiomatize the necessary properties, such as dynamic length and additional operations like concatenation and extraction.

We are interested in a variant of the theory of sequences, which we call the theory of n-indexed sequences. They differ from sequences mainly in their indexing, as they are not necessarily 0-indexed but n-indexed, as their name suggests. This means they 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 like Ada. Since there is no dedicated theory for such sequences, reasoning over them cannot be done 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 ways to reason over it using existing theories and by adapting calculi from the theory of sequences to the theory of n-indexed sequences.

Related work:

The SMT theory of sequences was introduced by Bjorner et al. [1]. Several contributions explored this theory, its syntax and semantics [2], and its decidability [3, 4].

Our theory of n-indexed sequences and the calculi we developed are based on the contribution by Sheng et al. [5], which in turn is based on reasoning about the theories of strings [6, 7] and arrays [8].

SMT 2024: 22nd International Workshop on Satisfiability Modulo Theories
hra687261@gmail.com (H. R. Ait El Hara); francois.bobot@ocamlpro.com (F. Bobot); gbury@gmail.com (G. Bury)
https://hra687261.github.io/ (H. R. Ait El Hara); https://gbury.eu/ (G. Bury)
0000-0001-7909-0413 (H. R. Ait El Hara); 0000-0002-6756-0788 (F. Bobot); 0009-0002-1267-251X (G. Bury)

Other contributions have extended the theory of arrays with properties that are present in sequences, such as length [9, 10, 11] and a concatenation function [12].

2. Notation

We refer to the theory of n-indexed sequences as the theory of n-sequences or the NSeq theory, and n-indexed sequence terms will be referred to as n-sequences, n-sequence terms, or NSeq terms. Their sort will be denoted as NSeq E, where E is the sort of the elements of the n-sequence. We will refer to the theory of sequences as the Seq theory. Int is the sort of integers from the theory of Linear Integer Arithmetic. $\min$ and $\max$ are the usual mathematical functions. ite is the ite SMT-LIB function; it takes a boolean expression and two expressions of the same sort and returns the first one if the boolean expression is true and the second otherwise. The $\mathrm{let}\ x = v, y$ symbol binds a variable x to a value v in a term y.

In the remainder of the paper, we will use s, s_n, w, w_n, y_1, y_2, z_1, and z_2 to represent NSeq terms, with n being a positive integer. k, k_1, k_2, and k_3 will represent fresh NSeq term variables. i and j will be used as integer index terms, and u and v will be used as NSeq element terms. We assume that all the terms we use are well-sorted.

3. The Theory of n-Indexed Sequences

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

SMT-LIB symbol	Sort
nseq.get	NSeq E → Int → E
nseq.set	NSeq E → Int → E → NSeq E
nseq.first	NSeq E → Int
nseq.last	NSeq E → Int
nseq.const	Int → Int → E → NSeq E
nseq.relocate	NSeq E → Int → NSeq E
nseq.concat	NSeq E → NSeq E → NSeq E
nseq.slice	NSeq E → Int → Int → NSeq E
nseq.update	NSeq E → NSeq E → NSeq E

We present in this section the theory of n-indexed sequences. The signature of the NSeq theory is presented in the table 1. In the remainder of the paper, when referring to the symbols of the theory, the prefix "nseq." of the symbols will be omitted.

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

- f_s: the first index of s.
- l_s: the last index of s.
- $\mathrm{get}(s, i)$: if the index i is within the bounds of s, returns the element associated with i in s; otherwise, returns an uninterpreted value. An uninterpreted value is one which is not constrained and can be any value of the right sort.
- $\mathrm{set}(s, i, v)$: if i is within the bounds of s, creates a new n-sequence in which i is associated with v, and the other indices within the bounds are associated with the same values as the corresponding indices in s; otherwise, returns s.
- $\mathrm{const}(f, l, v)$: creates an n-sequence with f as the first index and l as the last, and if it is not empty, all the indices within its bounds are associated with the value v.
- $\mathrm{relocate}(a, f)$: given an n-sequence a and an index f, returns a new n-sequence b which has f as its first index and $(f + \mathrm{l}_a - \mathrm{f}_a)$ as its last index, and associates with each index i within the bounds of b the same value associated with the index $(i - \mathrm{f}_b + \mathrm{f}_a)$ in a.

- $\mathrm{concat}(a, b)$: if a is empty, returns b; if b is empty, returns a; if $\mathrm{f}_b = \mathrm{l}_a + 1$, returns a new n-sequence in which the first index is f_a, the last index is l_b, and all the indices within the bounds of a are associated with the same values as the corresponding indices in a, as well as the indices within the bounds of b are associated with the same values as the corresponding indices in b; if $\mathrm{f}_b \neq \mathrm{l}_a + 1$, returns a.
- $\mathrm{slice}(a, f, l)$: if $\mathrm{f}_a \leq f \leq l \leq \mathrm{l}_a$, returns a new n-sequence for which the first index is f, the last index is l, and all the indices between f and l are associated with the same values as the corresponding indices in a; otherwise, returns a.
- $\mathrm{update}(a, b)$: if a is empty, or b is empty, or the property $\mathrm{f}_a \leq \mathrm{f}_b \leq \mathrm{l}_b \leq \mathrm{l}_a$ doesn't hold, returns a; otherwise, returns a new n-sequence c such that $\mathrm{f}_c = \mathrm{f}_a$ and $\mathrm{l}_c = \mathrm{l}_a$, and for all the indices within the bounds of c, if they are within the bounds of b, then they are associated with the same values as they are in b, otherwise they are associated with the values to which they are associated in a.

Definition 1 (Bounds). *The bounds of an n-sequence s are its first and last index, which are respectively denoted as f_s and l_s, and which correspond to the values returned by the functions nseq.first(s) and 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 (Extensionality). *The theory of n-indexed sequences is extensional, which means that n-sequences that contain the same elements are equal. Therefore, given two n-sequences a and b:*

$$\begin{gathered}\mathrm{f}_a = \mathrm{f}_b \wedge \mathrm{l}_a = \mathrm{l}_b \wedge \\ (\forall i : \mathit{Int}, \mathrm{f}_a \leq i \leq \mathrm{l}_a \rightarrow \mathrm{get}(a, i) = \mathrm{get}(b, i)) \\ \rightarrow a = b\end{gathered}$$

Definition 3 (Empty n-sequence). *An n-sequence s is said to be empty if $\mathrm{l}_s < \mathrm{f}_s$. Two empty n-sequences a and b are equal if $\mathrm{f}_a = \mathrm{f}_b$ and $\mathrm{l}_a = \mathrm{l}_b$; otherwise, they are distinct.*

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 follow the semantics of the corresponding symbols in the NSeq theory.

Another way is to use the theory of sequences and the theory of algebraic data types. It consists in 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 theory and Algebraic Data Types, is attached in Appendix A.1.

Although this approach allows us to reason over n-indexed sequences, it is not ideal to depend on two theories to do so. 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.

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. [5] on the Seq theory, where two calculi were proposed. The first is called the BASE calculus, based on a string theory calculus that reduces functions selecting and storing one element at an index to concatenations of sequences. The second is called the EXT calculus, and it handles these functions 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 syntax and semantics of many symbols:

- const and 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 a third argument corresponds to set in the NSeq theory, while seq.update with a sequence as a third argument corresponds to update in the NSeq theory, which takes only two NSeq terms 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 Seq 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 over the LIA (Linear Integer Arithmetic) theory is supported, and where unsatisfiability in one of the theories implies unsatisfiability of the entire reasoning. We will only present the rules that handle the symbols of the NSeq theory.

5.1. Common calculus

Definition 4 (Equivalence modulo relocation). *Given two NSeq terms s_1 and s_2, the terms are said to be equivalent modulo relocation, denoted with the equivalence relation $s_1 =_{reloc} s_2$, such that:*

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

Equivalence modulo relocation represents equivalence between n-sequences relative to their starting index. Two n-sequences are equivalent modulo relocation if they have the same set of elements in the same order, but start at different indices.

Definition 5 (NSeq term normal form). *For simplicity and consistency with Seq theory calculi, we introduce the concatenation operator $::$ with the invariant:*

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

This operator is used to normalize NSeq terms. It differs from concat *by not having to check the condition that $\mathrm{f}_{s_2} = \mathrm{l}_{s_1} + 1$ before concatenation, as it is ensured by the invariant.*

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

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

(1) and (2) remove empty NSeq terms from normal form. (3) and (4) make sure that when an NSeq term appear in the normal of another NSeq term 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 NSeq terms under appropriate conditions. If an NSeq term 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 equivalence modulo relocation.

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 a new normal form for the NSeq terms they operate on. In the R-Get rule, when i is within the bounds of s, a new normal form of s is introduced. It includes a constant NSeq term of size one at the ith position storing the value v, and two variables, k_1 and k_2, to represent the left and right segments of NSeq term 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, representing segments on the left and right of the ith index. s_1 has a constant NSeq term of size one holding the value v at the ith index, while s_2 introduces another variable, k_2, of the same size and position.

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

$$\text{NS-Slice}\ \frac{s_1 = slice(s, f, l)}{\begin{array}{l} (f < \text{f}_s \vee l < f \vee \text{l}_s < l) \wedge s_1 = s \quad || \\ \text{f}_s \leq f \leq l \leq \text{l}_s \wedge s = k_1 :: s_1 :: k_2 \end{array}}$$

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

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

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

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

Figure 1: Common inference rules for the NS-BASE and NS-EXT calculi

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

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

Figure 2: NS-BASE specific inference rules

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 how they are treated in the array decision procedure described in [8]. The Get-Intro rule introduces a get operation from a set operation. The Get-Set operation is equivalent to what is commonly referred to as the read-over-write or select-over-store rule in the Array theory, allowing the application of a get operation over a set operation. The Set-Bound rule ensures that a set operation is performed within the bounds of the target NSeq term, or that the resulting NSeq term is equivalent to the one it was applied on. The Get-Concat, Set-Concat, and Set-Concat-Inv rules illustrate how get and set operations are handled when applied to an NSeq term in normal form, where

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

$$\text{Set-Concat}\ \frac{s_1 = set(s_2,i,v) \qquad s_2 = w_1 :: ... :: w_n}{\begin{array}{c} i < \mathrm{f}_{s_2} \vee \mathrm{l}_{s_2} < i \quad || \\ s_1 = k_1 :: ... :: k_n \wedge \mathrm{f}_{w_1} \leq i \leq \mathrm{l}_{w_1} \wedge k_1 = set(w_1,i,v) \wedge \\ \mathrm{f}_{k_1} = \mathrm{f}_{w_1} \wedge \mathrm{l}_{k_1} = \mathrm{l}_{w_1} \wedge ... \wedge \mathrm{f}_{k_n} = \mathrm{f}_{w_n} \wedge \mathrm{l}_{k_n} = \mathrm{l}_{w_n} \quad || \quad ... \quad || \\ s_1 = k_1 :: ... :: k_n \wedge \mathrm{f}_{w_n} \leq i \leq \mathrm{l}_{w_n} \wedge k_n = set(w_n,i,v) \wedge \\ \mathrm{f}_{k_1} = \mathrm{f}_{w_1} \wedge \mathrm{l}_{k_1} = \mathrm{l}_{w_1} \wedge ... \wedge \mathrm{f}_{k_n} = \mathrm{f}_{w_n} \wedge \mathrm{l}_{k_n} = \mathrm{l}_{w_n} \end{array}}$$

$$\text{Set-Concat-Inv}\ \frac{s_1 = set(s_2,i,v) \qquad s_1 = w_1 :: ... :: w_n}{\begin{array}{c} i < \mathrm{f}_{s_2} \vee \mathrm{l}_{s_2} < i \quad || \\ s_2 = k_1 :: ... :: k_n \wedge \mathrm{f}_{w_1} \leq i \leq \mathrm{l}_{w_1} \wedge w_1 = set(k_1,i,v) \wedge \\ \mathrm{f}_{k_1} = \mathrm{f}_{w_1} \wedge \mathrm{l}_{k_1} = \mathrm{l}_{w_1} \wedge ... \wedge \mathrm{f}_{k_n} = \mathrm{f}_{w_n} \wedge \mathrm{l}_{k_n} = \mathrm{l}_{w_n} \quad || \quad ... \quad || \\ s_2 = k_1 :: ... :: k_n \wedge \mathrm{f}_{w_n} \leq i \leq \mathrm{l}_{w_n} \wedge w_n = set(k_n,i,v) \wedge \\ \mathrm{f}_{k_1} = \mathrm{f}_{w_1} \wedge \mathrm{l}_{k_1} = \mathrm{l}_{w_1} \wedge ... \wedge \mathrm{f}_{k_n} = \mathrm{f}_{w_n} \wedge \mathrm{l}_{k_n} = \mathrm{l}_{w_n} \end{array}}$$

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

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

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

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

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

Figure 3: NS-EXT specific inference rules

the operations affect the right component of the concatenation within its bounds. The Get-Const rule addresses the special case where a get operation is applied to a constant NSeq term. The Get-Reloc rule facilitates the propagation of constraints on index-associated values in NSeq terms from one term to others that are equivalent modulo relocation.

6. Implementation

We have implemented a prototype of the described calculi in the Colibri2 CP (Constraint Programming) solver. In this section, we discuss some of the implementation choices we made.

The rewriting rules described in Assumption 1 are applied whenever applicable using a callback system. When the conditions are satisfied, the corresponding rewriting rule is triggered.

Equivalence modulo relocation is managed using a disjoint-set (union-find) data structure. In this data structure, the elements of the sets are NSeq terms, and the equivalence relation is defined by $=_{reloc}$ as previously specified. By definition, if two elements of an equivalence class are at the same relocation

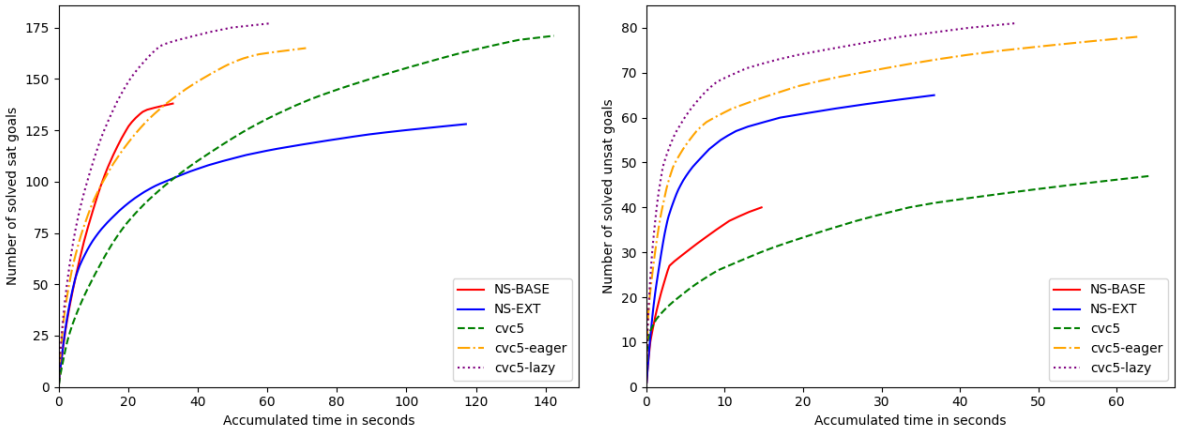

Figure 4: Number of solved goals by accumulated time in seconds on quantifier-free Seq benchmarks translated from the QF_AX SMT-LIB benchmarks

offset from the representative, they are equal. The data structure maintains, for each equivalence class, a mapping from offset to an element of the class that is at that specific relocation offset from the representative. This facilitates efficient detection of such equalities with minimal overhead.

7. Experimental Results

In this section, we present experimental results of the calculi described in the previous section. Currently, our experiments have focused exclusively on quantifier-free benchmarks that utilize only the theory of sequences and the theory of uninterpreted functions. These benchmarks constitute a subset of those employed in the paper [5], originally translated into the Seq theory from the QF_AX SMT-LIB benchmarks.

To achieve this, we implemented support for the Seq theory in our solver by translating Seq terms into NSeq terms. The translation process is as follows:

- Seq terms: NSeq terms for which the first index is 0 and the last index is greater or equal than -1.
- seq.empty: an NSeq term of the same sort, in which the first index is 0 and the last is -1, denoted ϵ.
- seq.unit(v): $\mathrm{const}(0, 0, v)$
- seq.len(s): $\mathrm{l}_s - \mathrm{f}_s + 1$
- seq.nth(s, i): $\mathrm{get}(s, i)$
- seq.update(s_1, i, s_2):

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

- seq.extract(s, i, j):

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

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

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

The figure 4 depicts the number of satisfiable and unsatisfiable goals solved over accumulated time using our prototype implementation and the cvc5 SMT solver with different command-line options. NS-BASE and NS-EXT refer to our implementations[1], described in section 5, which can be used by running Colibri2 with the command-line options `--nseq-base` and `--nseq-ext`, respectively. We compare our implementation with the Seq theory implementation in cvc5 (version 1.1.1). In the graphs in figure 4, cvc5 corresponds to running cvc5 with the command-line option `--strings-exp`, necessary for using cvc5's solver for the Seq theory. cvc5-eager uses the same option with `--seq-arrays=eager`, and cvc5-lazy with `--seq-arrays=lazy`, these options indicate different strategies for using an array-inspired solver for the Seq theory.

Examining the graph on the right, which shows performance on unsatisfiable goals, we observe that our NS-EXT implementation outperforms cvc5 in both time and number of goals solved. Meanwhile, NS-BASE initially solves more goals than cvc5, but solves fewer overall. Additionally, cvc5-eager and cvc5-lazy solve more goals in less time compared to the others. The same trends apply to the satisfiable case, with the exception that cvc5 also surpasses NS-EXT in both time and number of goals solved once the 20-second threshold is reached.

In our context, focused on program verification, the performance on unsatisfiable goals holds greater significance, though the satisfiable case remains useful. Since Colibri2 constructs concrete counterexamples before concluding satisfiability, we aim to enhance our current model generation technique for n-sequences. In the unsatisfiable case, while we compete closely with the state-of-the-art SMT solver cvc5, we have observed that some goals unsolved within a short timeout (5 seconds) also remain unsolved with longer timeouts, suggesting potential performance limitations in our propagators for the NSeq theory. It's also notable that our translation from Seq to NSeq in Colibri2 introduces more complex terms, and Colibri2 lacks clause learning, making decisions costlier than in other SMT solvers.

8. Conclusion

In this paper, we explored the topic of 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.

Looking ahead, our future work will delve deeper into different reasoning approaches for this theory, exploring their respective strengths and weaknesses through benchmarking with n-indexed sequences. We aim to prove the correctness of our developed calculi and explore alternative methods for reasoning over n-sequences beyond traditional sequence or string reasoning. Moreover, we seek to identify additional applications for this theory beyond programming languages where n-indexed sequences are present.

References

[1] N. Bjørner, V. Ganesh, R. Michel, M. Veanes, An SMT-LIB Format for Sequences and Regular Expressions, Strings (2012).

[2] H. R. Ait El Hara, F. Bobot, G. Bury, On SMT Theory Design: The Case of Sequences, in: Kalpa Publications in Computing, volume 18, EasyChair, 2024, pp. 14–29. URL: https://easychair.org/publications/paper/qdvJ. doi:10.29007/75t1, iSSN: 2515-1762.

[3] C. A. Furia, What's Decidable about Sequences?, in: A. Bouajjani, W.-N. Chin (Eds.), Automated Technology for Verification and Analysis, Springer, Berlin, Heidelberg, 2010, pp. 128–142. doi:10.1007/978-3-642-15643-4_11.

[4] A. Jeż, A. W. Lin, O. Markgraf, P. Rümmer, Decision Procedures for Sequence Theories, in: C. Enea, A. Lal (Eds.), Computer Aided Verification, Lecture Notes in Computer Science, Springer Nature Switzerland, Cham, 2023, pp. 18–40. doi:10.1007/978-3-031-37703-7_2.

[1] Available at: https://git.frama-c.com/pub/colibrics/-/tree/smt2024 (commit SHA: 43024e674ef26673d2495f3b186954fa37bc3890)

[5] Y. Sheng, A. Nötzli, A. Reynolds, Y. Zohar, D. Dill, W. Grieskamp, J. Park, S. Qadeer, C. Barrett, C. Tinelli, Reasoning About Vectors: Satisfiability Modulo a Theory of Sequences, Journal of Automated Reasoning 67 (2023) 32. URL: https://doi.org/10.1007/s10817-023-09682-2. doi:10.1007/s10817-023-09682-2.

[6] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, M. Deters, A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions, volume 8559, Springer International Publishing, Cham, 2014, pp. 646–662. URL: http://link.springer.com/10.1007/978-3-319-08867-9_43. doi:10.1007/978-3-319-08867-9_43, book Title: Computer Aided Verification Series Title: Lecture Notes in Computer Science.

[7] M. Berzish, V. Ganesh, Y. Zheng, Z3str3: a string solver with theory-aware heuristics, in: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design, FMCAD '17, FMCAD Inc, Austin, Texas, 2017, pp. 55–59.

[8] J. Christ, J. Hoenicke, Weakly Equivalent Arrays, in: C. Lutz, S. Ranise (Eds.), Frontiers of Combining Systems, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 119–134. doi:10.1007/978-3-319-24246-0_8.

[9] M. P. Bonacina, S. Graham-Lengrand, N. Shankar, CDSAT for Nondisjoint Theories with Shared Predicates: Arrays With Abstract Length, Satisfiability Modulo Theories workshop, CEUR Workshop Proceedings 3185 (2022). URL: https://par.nsf.gov/biblio/10358980-cdsat-nondisjoint-theories-shared-predicates-arrays-abstract-length.

[10] S. Ghilardi, A. Gianola, D. Kapur, C. Naso, Interpolation Results for Arrays with Length and MaxDiff, ACM Transactions on Computational Logic 24 (2023) 28:1–28:33. URL: https://doi.org/10.1145/3587161. doi:10.1145/3587161.

[11] A. R. Bradley, Z. Manna, H. B. Sipma, What's Decidable About Arrays?, in: E. A. Emerson, K. S. Namjoshi (Eds.), Verification, Model Checking, and Abstract Interpretation, Springer, Berlin, Heidelberg, 2006, pp. 427–442. doi:10.1007/11609773_28.

[12] Q. Wang, A. W. Appel, A Solver for Arrays with Concatenation, Journal of Automated Reasoning 67 (2023) 4. URL: https://doi.org/10.1007/s10817-022-09654-y. doi:10.1007/s10817-022-09654-y.

A. Appendix

A.1. 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)))
```