



On SMT Theory Design: The Case of Sequences

Hichem Rami Ait El Hara^{1,2}, François Bobot², and Guillaume Bury¹

¹ OCamlPro, Paris, France

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

Abstract

Choices in the semantics and the signature of a theory are integral in determining how the theory is used and how challenging it is to reason over it. Our interest in this paper lies in the SMT theory of sequences. Various versions of it exist in the literature and in state-of-the-art SMT solvers, but it has not yet been standardized in the SMT-LIB. We reflect on its existing variants, and we define a set of theory design criteria to help determine what makes one variant of a theory better than another. The criteria we define can be used to appraise theory proposals for other theories as well. Based on these criteria, we propose a series of changes to the SMT theory of sequences as a contribution to the discussion regarding its standardization.

1 Introduction

The SMT theory of arrays, introduced by McCarthy [19], maps index terms of a given sort to value terms of another sort. The theory provides two functions: the `select(a, i)` function, which takes an array a and index i , and returns the value stored at i in a ; and the `store(a, i, v)` function, which takes an array a , an index i , and a value v , and returns a modified copy of a in which the value v is at the index i . This theory is minimal and generic, and many efficient decision procedures have been proposed for it [11, 8, 10]. However, the theory’s lack of expressiveness hinders reasoning on more complex data structures, as it makes it necessary to define additional functions and to use axiomatization, eventually with quantifiers, to express properties on such data structures.

Thus, when it comes to verifying properties of a given data structure using SMT solvers, it is more convenient to have a tailored theory that clearly and concisely describes the semantics of the higher-level operations on that data structure. Not only does this make verification easier for the user, but it can also pave the way for more dedicated and efficient decision procedures for the theory. Examples of such theories are the theory of strings and the theory of sequences, which have both sparked a lot of interest in recent years.

Sequences are a common data structure in programming languages, although they may be known by different names and have various implementations. Sequences can have fixed sizes, like arrays in C, C++, Rust, OCaml, and Java, or they can be dynamic, like vectors in C++ and Rust, ArrayLists in Java, arrays in JavaScript, and lists in Python. In addition to the common array operations, such as storing and selecting values at an index, some languages support higher-level operations such as concatenation, slicing, mapping, filtering, and folding.

Recent studies tend to confirm our hypothesis: it is more efficient to verify properties of such data structures with an SMT solver by using the theory of sequences rather than the theory of arrays with additional axiomatization [22].

The theory of sequences differs from the theory of arrays in that sequences are dynamic and can change in size, while arrays have a fixed size determined by the sort of the indices and the number of possible values it has. Sequences are always indexed by integers, whereas arrays don't have such restrictions on their index sort. Additionally, the signature of the theory of sequences is richer than that of the theory of arrays; it includes functions for concatenation, slicing, subsequence extraction, etc. The $\text{nth}(s, i)$ function from the theory of sequences takes a sequence s and an index i , returning the value stored at the i th index of the sequence, akin to the `select` function in the array theory. However, the mathematical interpretation of this function on sequences is partial to valid indices, which are those within the bounds of the sequence. As the SMT-LIB is a total logic, such functions are totalized.

A function is considered partial when it is not defined for all possible inputs, thus being applicable outside its domain. The returned value in such cases depends on how the function is totalized. Partial function totalization can be achieved in three ways. Firstly, through underspecification, which consists in returning an uninterpreted value. An uninterpreted value has no associated interpretation, meaning it is unconstrained and can be any value of the right sort. Secondly, overspecification, which entails returning a default constant value when a partial function is applied outside its domain. Thirdly, the additional argument approach, where an argument is added to the function. This argument will be returned when the function is applied outside its domain, allowing the user to determine the return value by providing that value to the function.

In this paper, we aim to discuss the design of the theory of sequences. We will define a set of criteria that ought to be taken into consideration when designing an SMT theory. Additionally, we will describe the various approaches used in the literature and in SMT solvers to totalize partial functions. Then, we will apply the criteria we have defined to suggest some variations to the theory of sequences. These variations aim to improve the theory for both those trying to reason over it and the users of the theory.

The paper is organized as follows: We begin by stating the notation we use in Section 2. Section 3 summarizes the known theories of sequences in the literature and in SMT solvers, and discusses the differences between them. In Section 4, we define a set of SMT theory design criteria that we consider as important to take into account when designing an SMT theory. Section 5 explores how partial functions are dealt with in SMT literature and discusses which approach is adequate when. In Section 6, we propose variations of the theory of sequences that we have designed, taking into account the criteria described in Section 4, and discuss why we believe they are better. Finally, we conclude in Section 7.

Related work Regarding theory design, some standardized theories in the SMT-LIB are actually based on published work proposing signatures and semantics for these theories. It is the case for the theory of Floating Point arithmetic [20, 7] and the theory of strings [3]. To our knowledge, theory design is not discussed in other contributions as clearly as it is here, but there are extensive discussions on the subject available on the SMT-LIB mailing list¹.

On the subject of handling partial functions, it is a known problem in mathematical logic in general. There are few works on it in Satisfiability Modulo Theories [2]. However, there is more significant literature on the topic in related fields such as HOL (Higher Order Logic) and proof assistant development [21, 17, 16].

¹Available at: <https://groups.google.com/g/smt-lib>

The theory of sequences was introduced relatively recently, it was first formalized by Bjørner et al. [3]. Sheng et al. developed calculi to reason over their variant of the theory of sequences [22]. Additionally, there are works on the theory of arrays that extend the theory with a length function [5, 15], a concatenation function [23], and others [6, 12, 14], giving arrays similar properties to those of sequences.

2 Notation

`Array` is the sort of arrays, `Elem` is the sort of values, `Int` is the sort of integers, `Seq` is the sort of sequences, and `Bool` is the sort of boolean values. The symbols `min` and `max` denote the usual mathematical functions. `ite` is the commonly used function in the SMT-LIB Standard. The symbol α represents a sort variable and can be any sort. The sort of higher-order functions $(T_1 \rightarrow \dots \rightarrow T_{n-1} \rightarrow T_n)$ is used as a simplification for the sort `ArrayT1(Array...(ArrayTn-1 Tn))`. In let $x = v, y$, the let symbol binds a variable x to a term v in a term y .

3 The theory of sequences

Sequences are 0-indexed ordered collections of elements with dynamic lengths. The SMT theory of sequences was proposed by Bjørner et al. [3] as a generalization of the theory of strings to non-character values (`Seq`). State-of-the-art SMT solvers such as CVC5² [22] and Z3³ support theories of sequences, referred to as `Seqcvc5` and `Seqz3` respectively. Their signatures share many symbols with `Seq`, along with some additions and deductions.

Extensions to the theory of arrays, such as Wang and Appel’s theory of arrays with concatenation [23], referred to as `Arrayc`, extend the theory of arrays with length, slice, and concat operators. Since this provides arrays with properties similar to those of sequences, mainly 0-indexing and length, we will refer to `Arrayc`’s arrays as sequences.

3.1 Known theories

We present the signatures of `Seqcvc5` and `Seqz3`, starting with the shared symbols of the two, then the specificities of each one:

- Common symbols to `Seqcvc5` and `Seqz3`:
 - `seq.empty`: the empty sequence
 - `seq.unit(v)`: a sequence of length 1 containing only the value v
 - `seq.len(s)`: the length of the sequence s
 - `seq.nth(s, i)`: the value associated with the i th index of s if i is within the bounds of s , an uninterpreted value otherwise
 - `seq.extract(s, i, l)`: the extracted maximal sub-sequence of s , starting at i of length l if i is within the bounds of s and l is positive, the empty sequence otherwise
 - `seq.concat(s_1, \dots, s_n)`: the concatenation of the sequences s_1, \dots , and s_n
 - `seq.at(s, i)`: a unit sequence containing the i th value in s if i is within the bounds of s , the empty sequence otherwise

²CVC5’s sequence theory: <https://cvc5.github.io/docs-ci/docs-main/theories/sequences.html>

³Z3’s sequence theory: <https://microsoft.github.io/z3guide/docs/theories/Sequences>

- `seq.contains(s_1, s_2)`: true if s_1 is a sub-sequence of s_2 , false otherwise
- `seq.indexof(s, s', i)`: the first position of s' in s at or after i , -1 if there are no occurrences
- `seq.replace(s, s_1, s_2)`: the resulting sequence from replacing the first occurrence of s_1 with s_2 in s if s_1 occurs in s , s otherwise
- `seq.prefixof(s', s)`: true if s' is a prefix of s , false otherwise
- `seq.suffixof(s', s)`: true if s' is a suffix of s , false otherwise
- Symbols belonging to `Seqcvc5` only:
 - `seq.replace_all(s, s_1, s_2)`: the resulting sequence from replacing all occurrences of s_1 with s_2 in s , s if s_1 does not occur in s
 - `seq.rev(s)`: the resulting sequence from reversing s
 - `seq.update(s_1, i, s_2)`: New sequence of the same size as s_1 , in which, if i is within the bounds of s_1 , then the values from i to $i + \text{seq.len}(s_2) - 1$ are the same values as in s_2 and the other values are the same as in s_1 , otherwise it equals s_1
- Symbols belonging to `Seqz3` only:
 - `seq.map(f_n, s)`: the sequence of sort (`Seq E'`) resulting from applying f_n , which is of sort (`Array E E'`) with `E` as the sort of the values of s , to all the values of s
 - `seq.mapi(f_n, o, s)`: the sequence of sort (`Seq E'`) resulting from applying f_n , which is of sort (`Array Int (Array E E')`) with `E` as the sort of the values of s , to all the values of s and their indices starting from the offset o
 - `seq.fold_left(f_n, b, s)`: The result of folding over s of sort `Seq E`, with an initial value b of sort `E'`, using the function f_n of sort (`Array E' (Array E E')`)
 - `seq.fold_lefti(f_n, o, b, s)`: The result of folding over the values of s of sort `Seq E` and their indices, with an initial value b of sort `E'`, using the function f_n of sort (`Array Int (Array E' (Array E E'))`), starting from the offset o

The `seq.update` function is described differently in the paper that describes the reasoning implemented in CVC5 [22] for the theory of sequences and in the documentation of CVC5⁴. In the paper, it is described as a function that sets only the value of one index, and takes that value as a third argument, while in the documentation it takes a sequence as a third argument.

The signature of `Arrayc` is the following:

- `lengthS(s)`: the length of s
- `nthS(i, s)`: the i th value of s , if i is out of the bounds of s then the value is the default value of the sort of values, the theory assumes that every value sort has a variable corresponding to the default value of that sort
- `repeatS(v, n)`: a sequence of size n if n is positive, in which all values are v , the empty sequence if n is negative
- `appS(s_1, s_2)`: concatenates s_1 and s_2

⁴CVC5's sequence theory: <https://cvc5.github.io/docs-ci/docs-main/theories/sequences.html>

- $\text{slice}_S(i, j, s)$: a sub-sequence of s from $\max(i, 0)$ to $\min(j, l)$, with l as the length of s , the empty sequence if such a sub-sequence does not exist
- $\text{map}_f(s_1, \dots, s_k)$: the sequence resulting from applying f element-wise to the n first elements of the sequences s_1, \dots, s_k , where n is $\min(\text{length}_S(s_1), \dots, \text{length}_S(s_k))$
- $\text{update}(i, s, x)$: returns an updated version of s in which i is mapped to x if i is within the bounds of s . It is mentioned that the function `update` is reduced to a concatenation of the sequences `slice(0, i, s)`, `repeat(v, 1)` and `slice(i + 1, length(s), s)` when i is within the bounds of s .

The map_f symbol from Array_c is similar to the `map` function over arrays described in a paper by de Moura and Bjørner presenting the CAL (Combinatory Array Logic) array decision procedure [11].

3.2 Discussion

The theory of sequences is seen as a generalization of the theory of strings to non-character values. Its signature is largely based on that of the theory of strings. From the available literature [22] and the properties of sequences in the theory, such as being 0-indexed and having dynamic lengths, we understand that the theory of sequences serves the purpose of more adequately representing arrays as found in programming languages than what the theory of arrays can do. However, a `seq.set` function in `Seq`, which would correspond to the `store` function in the theory of arrays and store one value at one index in a sequence, is missing. This is problematic as that function is usually present in array-like data structures, such as the array assignment operation in C.

Having a rich signature that allows all necessary functions and predicates on values of a theory makes that theory expressive. This means that when users need to express properties in that theory, they will not have to define or axiomatize them if they can rely on the theory's built-in symbols. However, a theory's signature affects how to reason over it. For example, when working with `Seq`, it is natural to adapt reasoning over strings to reasoning over sequences, since the signatures of the theory of strings and that of `Seq` have many similarities. If only a fragment of `Seq` is used, for example, one in which operations over sub-sequences such as `seq.extract`, `seq.concat`, and `seq.update` are not supported, then reasoning over arrays can more easily be adapted to that fragment of `Seq` than to `Seq` itself. Therefore, having simpler, minimal fragments of a theory can allow for the development of tailored reasoning for that fragment, which can be more efficient than the reasoning used for the whole theory. This is, for example, the case for Integer (resp. Real) Difference Logic, which is a fragment of Integer (resp. Real) Linear Arithmetic and has its own reasoning techniques.

Looking at the Seq_{z3} and Array_c theories, the `map` symbol is present, albeit with different semantics. Seq_{z3} has a `seq.map` function and a `seq.fold_left` function similar to those commonly encountered in ML and more generally in functional programming languages. The `seq.map` function applies a function to all the elements of a sequence and produces a new sequence, while `seq.fold_left` iterates over all the elements of a sequence and applies a function to them and to an accumulator that can be of any other sort, returning that accumulator to pass it to the function in the next iterations. Since the SMT-LIB standard version 2.6 does not support higher-order functions, the functions are represented using arrays, and to express complex functions using arrays, it is usually necessary to use quantifiers, which SMT solvers often struggle with. However, such functions would be useful for modeling array-like or list-like data structures from functional programming languages. The map_f function is similar to the one

present in the CAL extension of the theory of arrays [11], and it applies a function f element-wise to the k elements of n arrays, where k is the length of the smallest of the n arrays, so applying it to one array makes it similar to the `seq.map` function from `Seqz3`.

4 Design Criteria

In this section, we will define a set of criteria that we consider important when designing an SMT theory and choosing its signature and semantics.

4.1 Expressiveness

Having clear semantics and a rich signature with both the necessary functions to perform commonly encountered operations and the predicates to express the properties that need to be verified is necessary in a theory. It simplifies the work of the users of that theory as they don't have to define or axiomatize additional symbols in the signature of the theory to use it. For example, in the theory of fixed-size bit-vectors, there are bitwise operations like `bvand` and `bvor`, as well as bit-vector arithmetic operations like `bvadd` and `bvsub`, which are commonly used when working with machine integers.

Written SMT formulae in files, whether by a user or automatically generated by tools like Why3 [13], Dafny [18], etc., can easily become large and complex. Therefore, expressiveness in theories plays an important role in clarity, understanding, and efficiency. Frequently, when a theory is missing needed features that can be built-in, adding them requires axiomatization with quantifiers, and quantifiers are known to be hard for solvers to work with and can be harder for users to understand.

4.2 Implementability and efficiency

The goal of this criterion is to ensure that reasoning over the theory in question is reasonably implementable in an SMT theory combination framework, with the constraints that come with it. Although the implementability and efficiency of a reasoning over a theory depend more on the reasoning itself rather than on the theory, they are significantly affected by the design choices of a theory, as that determines how complex the reasoning would need to be. Having a theory with a concise set of symbols and clear semantics will make it easier to formalize reasoning over it and to implement it. Additionally, theory functions should have well-formalized behavior, with not many special cases in which the behavior significantly changes or becomes more complicated.

This criterion is hard to measure, but when it comes to theory design, it is something that should be taken into account when deciding what signature and semantics to give a theory. It also involves a compromise with expressiveness since making a theory too expressive can complicate the task for those who try to reason over it and implement the reasoning.

4.3 Avoiding surprises and user-friendliness

A theory's semantics need to be as clear as possible. The symbols defined in the theory should have straightforward semantics and not many special cases in which they behave differently. Such complexity would make it harder to understand and solve any issues that may be encountered while using the theory. The symbols should also be consistent with one another in the theory, for example regarding associativity (when it is possible to choose) or the order of the

arguments. This consistency is evident in the theory of arrays, where both the `select` and `store` functions take an array term and an index term as the first and second arguments, respectively.

5 Handling of partial functions

Partial functions are functions that can be applied outside their domain of definition. To address partial functions, there are three known approaches: underspecification, overspecification, or by adding an argument to be returned when the function is applied outside its domain. In this section, we will clarify these three techniques.

5.1 Underspecification

This approach consists of treating values returned by functions when they are applied to arguments for which they are not defined as uninterpreted values. An uninterpreted value is unconstrained and can be any value of the correct sort. Therefore, when checking the satisfiability of an assertion containing such a value, the assertion is either satisfiable, or if it isn't, then it needs to be unsatisfiable for any possible value for that uninterpreted value. This approach was taken for integer and real division. When an integer (resp. real) value is divided by zero in the (Non-)Linear Integer (resp. Real) Arithmetic theory, the resulting value is uninterpreted.

When a user models a program's behavior, the underspecified approach is a safe choice because when a goal is proven to be unsatisfiable, it is proven for any value of the same sort. However, this approach can affect the decidability of a theory. For example, the theory of Real Difference Logic is decidable but becomes undecidable when combined with uninterpreted unary predicates [4].

Implementing this approach is not difficult if the framework in which it is implemented has a solver for the theory of uninterpreted functions. Otherwise, dependency on another theory can be problematic for developers of solvers that do not support the theory of uninterpreted functions. Moreover, in the satisfiable case, checking the validity of a model may require the value chosen by the solvers for the uninterpreted value that led to the satisfiable result. Yet, there is no syntax for specifying such values in the SMT-LIB standard version 2.6 [9].

5.2 Overspecification

Another way to handle partially defined functions is by selecting a constant value to be returned when the function is applied outside its domain. For instance, in the theory of Fixed Size Bit-Vectors, the function `bvudiv` takes two bit-vectors of the same size as arguments and returns a bit-vector of the same size, representing the result of the unsigned division between the two bit-vectors. When the second argument is a bit-vector of zeros, `bvudiv` returns a bit-vector representing the value -1 , which contains only ones. The rationale behind this choice is that bit-vectors are commonly used in circuit calculations, and when a circuit receives a zero, it returns all ones⁵. While such justifications apply to some choices of values, it is often unclear which value should be chosen, especially when the chosen value can be obtained by applying the function within its domain.

In the case of the Floating-Point Arithmetic theory, which follows the IEEE standard 754-2008 [1], the NaN value is used to represent undefined values. It serves as a catch-all case for the undefined behavior of the theory's functions. However, equivalent values do not exist in other theories. A clear advantage of this approach is that it facilitates the detection of undefined

⁵According to: <https://cs.nyu.edu/pipermail/smt-lib/2015/000966.html>

behavior since such cases are not silently treated. Encountering a NaN typically indicates that a partial function was applied outside its domain, although it's important to note that comparing a value to NaN always results in false, so the approach is not entirely foolproof and requires additional checks that values are not NaN.

The primary advantage of this approach is to have a predetermined value as a result whenever the function is applied outside its domain. This provides solvers with less flexibility in how to handle such values, establishing a sort of uniformity in their behavior in such cases. Additionally, it simplifies implementations, as solvers do not need to support the theory of uninterpreted functions or depend on it to handle such cases, they simply need to return the predetermined constant value.

Another example of this approach is seen in the `Arrayc` theory with the `nthS` function, which returns a default value when the index is out of bounds of the sequence. The theory assumes that every theory of values has such a default value. However, relying on a default value to be returned can lead to unexpected results. For instance, if applied on integer division, while $\frac{1}{0} = 2$ would not be provable, $\frac{1}{0} = \frac{2}{0}$ would be provable since $\frac{1}{0}$ and $\frac{2}{0}$ have the same integer default value.

5.3 Returned value as an argument

The idea behind this approach is to let the user decide. Instead of choosing a default value to return or returning an uninterpreted value, the user can choose which value to return by adding an argument to the function, the value of which will be returned when the function is applied outside its domain.

With this approach, if a user needs a partial function to return a specific constant value, they simply need to provide that constant value to the function. If they need an unconstrained value, they can also provide an uninterpreted constant. Implementation-wise, it is straightforward since it involves returning a value provided as an argument whenever the function is applied outside its domain. Moreover, it enhances user-friendliness by giving users control over the returned value when undefined behavior occurs, enabling easy detection and appropriate handling of such cases.

Although not common in the SMT-LIB standard, this approach has been suggested in past discussions on SMT theory design as a preferable solution compared to the previous two approaches, as it represents a compromise between them.

Considering a modified version of the `seq.nth` function from `Seq`, called `seq.nth'`, of sort `Seq → Int → Elem → Elem` where this approach is utilized, with the third argument representing the default value. If `seq.nth'` is used in a quantification, and underspecification behavior is desired, one way to achieve it is by defining a function `nth_defval` of sort `Seq → Int → Elem` such that `seq.nth(s, i) = seq.nth'(s, i, nth_defval(s, i))`. However, defining `nth_defval` is currently not possible in SMT-LIB version 2.6 due to the lack of polymorphism. Therefore, the usage of such an approach may be limited until SMT-LIB version 3 is released, which would allow polymorphism, already supported by some SMT solvers such as Alt-Ergo.

6 Designing theories of sequences

In this section, we propose changes to the theory of sequences by considering the criteria outlined in the previous sections. Our version of the theory is based on `Seqcvc5`, `Seqz3`, and `Arrayc`.

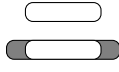
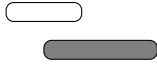
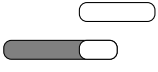
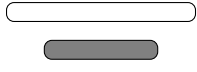

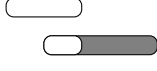

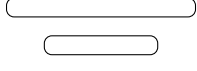
	no overflow	left overflow	right overflow	left-right overflow
Seq _{cvc5}				
Proposal				

Figure 1: Comparison of the semantics of update in Seq_{cvc5} and our proposal: the gray sequence is updated using the white sequence at different offsets.

6.1 Proposed changes to symbols of the theory of sequences

Below, we list and explain our modification choices for some sequence functions:

- **seq.get(s, i):** It is similar to the **select** function of the **Array** theory and the **seq.nth** (resp. **nth_S**) function from the **Seq_{cvc5}** and **Seq_{z3}** theories (resp. **Array_c** theory). It returns the value associated with the index i in the sequence s when i is within the bounds of s . The function is partial as it is not defined when the index i is out of the bounds of the sequence s . In the **Seq_{cvc5}** and **Seq_{z3}** theories, **seq.nth** is totalized by underspecification. While in **Array_c**, the default value approach is chosen, by assuming that all value theories have a variable that corresponds to that default value.

Due to the issues mentioned in the previous section related to the usage of the default argument as a return value for undefined behavior without polymorphism, we chose to follow the underspecification approach by returning an uninterpreted value when i isn't in the bounds of s .

The name of the symbol is **seq.get** and not **seq.nth** to make it consistent with the **seq.set** symbol that will be described next.

- **seq.set(s, i, v):** It is similar to the **store** **Array** theory function and not present in the theories of sequences, but it can be represented in **Seq_{cvc5}** with **seq.update($s, i, \text{seq.unit}(v)$)**. It is also similar to the **update** function of **Array_c**. It returns a new sequence in which the value v is stored at the i th index if i is within the bounds of s , and it is undefined otherwise.

In programming languages, accessing an array-like data structure out of its bounds usually results in an error. It can be argued that the function should show that it failed when it is applied outside its domain. One possible way to represent that failure is by returning the empty sequence. However, the semantics of the theory do not necessarily need to exactly follow the semantics of the programming languages they are used to prove the soundness of. Additionally, to use axioms from the **Array** theory decision procedures to reason over sequences, such as the **select-over-store** [10, 11] which dictates:

$$\text{select-over-store} \frac{a = \text{store}(b, i, v) \quad w = \text{select}(a, j) \quad a, b : \text{Array}, i, j : \text{Int}, v, w : \text{Elem}}{(i = j \wedge v = w) \vee (i \neq j \wedge \text{select}(a, j) = \text{select}(b, j))}$$

A similar axiom in the theory of sequences would state that setting the value of an index only changes the value associated with that index if it is within the bounds of the sequence and does not affect the other indices. Therefore, to support such axioms, returning s when

i is out of the bounds of i seems like a better choice, and it is the chosen approach for this function. `seq.set` is then defined by the following axiom:

$$\begin{aligned} s_2 = \text{seq.set}(s_1, i, v) &\equiv \\ \text{seq.len}(s_2) = \text{seq.len}(s_1) \wedge \\ \forall j : \text{Int}, 0 \leq j < \text{seq.len}(s_1) &\implies \text{seq.get}(s_2, j) = \text{ite}(j = i, v, \text{seq.get}(s_1, j)) \end{aligned}$$

- `seq.const`(l, v): equivalent to `repeat` in `Arrayc`, it is defined by the following axiom:

$$\begin{aligned} s = \text{seq.const}(l, v) &\equiv \\ \text{ite}(l \leq 0, s = \text{seq.empty}, \\ \text{seq.len}(s) = l \wedge \forall i : \text{Int}, 0 \leq i < l &\implies \text{seq.get}(s, i) = v) \end{aligned}$$

- `seq.slice`(s, i, j): Similarly to the `seq.extract` function from the `Seq`, `Seqcvc5`, and `Seqz3` theories and the `sliceS` function in the `Arrayc` theory. The purpose of this function is to make it possible to extract a subsequence from a sequence. It takes the target sequence argument s and two Integer arguments i and j . In the `Seqcvc5` and `Seqz3` theories, the choice was made for i to be the first index of the subsequence and j to be the length of the subsequence, likely to stay consistent with the substring extraction `str.substr` in the string theory. While the `sliceS` returns the subsequence from index i to index $j - 1$.

Each one of the two versions, first index and length, and first index and last index, can be expressed by the other one. It is unclear if one is better than the other. An advantage of having the length as an argument is that it is not needed to compute it to set the length constraint. On the other hand, if the reasoning can more easily get the slice of the sequence using the first and last index, then it might be better to have them as arguments instead of having to compute the last index.

In our case, we prefer a variation of the second version, one that is similar to the `extract` function from the Fixed-Size Bitvector theory and extracts the subsequence from the index i to the index j , because it seems more natural to reason over a slice of a sequence from its first index to its last index than with its first index and its length, but that is pretty arbitrary and as said before, they both can express one another.

Given the first and last indices i and j , the `seq.slice` function is partial since it is defined only when $0 \leq i \leq j < \text{seq.len}(s)$. That gives it four special cases to deal with:

- The negative length slice case, when $j < i$
- The left overflow case, when $i < 0$
- The right overflow case, when $\text{seq.len}(s) \leq j$
- The left-right overflow case, when both $i < 0$ and $\text{seq.len}(s) \leq j$ are true

In the case of the negative length slice, the natural solution would be to return an empty sequence since we are effectively trying to get a slice of non-strictly positive length. For the overflow cases, the suggested solution in the `Arrayc` theory seems best as it introduces consistency in how they are handled. It consists in taking the maximal subsequence of s that can be obtained between i and j , by selecting as the first index of the resulting slice $\max(i, 0)$ and as the last index $\min(j, \text{seq.len}(s) - 1)$. By following that choice, the

resulting axiom of the function is:

$$\begin{aligned}
s_2 &= \text{seq.slice}(s_1, i, j) \equiv \\
&\text{ite}(i \leq j, \text{let } i' = \max(i, 0), \text{let } j' = \min(j, \text{seq.len}(s_1) - 1), \\
&\text{seq.len}(s_2) = j' - i' + 1 \wedge \forall k : \text{Int}, i' \leq k \leq j' \rightarrow \text{seq.get}(s_2, k - i') = \text{seq.get}(s_1, k), \\
&s_2 = \text{seq.empty})
\end{aligned}$$

- $\text{seq.update}(s_1, i, s_2)$: Like the seq.update function from Seq_{cvc5} , which we will refer to as $\text{seq.update}_{\text{cvc5}}$. It updates a sequence s_1 starting from the index i with the sequence s_2 . It is only defined when $0 \leq i < i + \text{seq.len}(s_2) \leq \text{seq.len}(s_1)$ and has similar special cases to the seq.slice function:
 - The empty sequence case when $\text{seq.len}(s_2) = 0$
 - The left overflow case when $i < 0 \leq i + \text{seq.len}(s_2)$
 - The right overflow case when $\text{seq.len}(s_1) \leq i + \text{seq.len}(s_2)$
 - The left-right overflow case when both overflow conditions are true

The choice of the Seq_{cvc5} theory is to have $\text{seq.update}_{\text{cvc5}}$ behave as an iteration of seq.set on s_1 that goes from i to $i + \text{seq.len}(s_2) - 1$, writing the values of s_2 , and the iteration stops when i is not in the bounds of s_1 . So in the case of left overflow, the returned value is s_1 , in the case of right overflow, an intersection is done between s_1 and s_2 if $i < \text{seq.len}(s_1)$; otherwise, s_1 is returned. s_1 is also returned when s_2 is empty. The $\text{seq.update}_{\text{cvc5}}$ function's axiom is:

$$\begin{aligned}
s &= \text{update}_{\text{cvc5}}(s_1, i, s_2) \equiv \\
&\text{seq.len}(s) = \text{seq.len}(s_1) \wedge \\
&\text{ite}(0 \leq i < \text{seq.len}(s_1), \\
&\forall j : \text{Int}, 0 \leq j < \text{seq.len}(s_1) \implies \\
&\text{seq.get}(s, j) = \text{ite}(i \leq j < i + \text{seq.len}(s_2), \text{seq.get}(s_2, j - i), \text{seq.get}(s_1, j)), \\
&s = s_1)
\end{aligned}$$

In our case, we see seq.update more like a bag of seq.set operations, and we think that it would be simpler for it to respect the following axiom, which removes the condition that i needs to be in the bounds of s_1 :

$$\begin{aligned}
s &= \text{seq.update}(s_1, i, s_2) \equiv \\
&\text{seq.len}(s) = \text{seq.len}(s_1) \wedge \\
&\forall j : \text{Int}, 0 \leq j < \text{seq.len}(s_1) \implies \\
&\text{seq.get}(s, j) = \text{ite}(i \leq j < i + \text{seq.len}(s_2), \text{seq.get}(s_2, j - i), \text{seq.get}(s_1, j))
\end{aligned}$$

In addition, to follow the least surprise criteria, this also makes it behave consistently with seq.slice on how the left, right, and left-right overflow cases are treated, which is by taking the intersection between s_1 and s_2 offset to the i th index. Figure 1 illustrates the difference.

- $\text{seq.map}(f, s_1, \dots, s_n)$: The semantics for seq.map are the same as the semantics of the map_f in Array_c , as it is an n-ary version of the seq.map function in Seq_{z3} :

$$\begin{aligned} s &= \text{seq.map}(f, s_1, \dots, s_n) \equiv \\ &\text{seq.len}(s) = k \wedge \\ &\text{seq.get}(s, 0) = f(\text{seq.get}(s_1, 0), \dots, \text{seq.get}(s_n, 0)) \wedge \dots \wedge \\ &\text{seq.get}(s, k - 1) = f(\text{seq.get}(s_1, k - 1), \dots, \text{seq.get}(s_n, k - 1)) \end{aligned}$$

Where k is the length of the shortest of the sequences s_1 to s_n .

- $\text{seq.mapi}(f, o, s_1, \dots, s_n)$: A version of seq.map in which the mapped function is applied starting from an offset o :

$$\begin{aligned} s &= \text{seq.mapi}(f, o, s_1, \dots, s_n) \equiv \\ &\text{ite}(o \geq k, s = \text{seq.empty}, \\ &\text{seq.len}(s) = k - o \wedge \\ &\text{seq.get}(s, 0) = f(o, \text{seq.get}(s_1, o), \dots, \text{seq.get}(s_n, o)) \wedge \dots \wedge \\ &\text{seq.get}(s, k - o) = f(k - 1, \text{seq.get}(s_1, k - 1), \dots, \text{seq.get}(s_n, k - 1)) \end{aligned}$$

Where k is the length of the shortest of the sequences s_1 to s_n . In fact, it can be reduced to $\text{seq.map}(f, s_o, s_1, \dots, s_n)$ where s_o is a sequence of integers of size $k - o$, containing values going from o to $k - 1$, when k is greater than o and o is positive.

6.2 Theory of sequences proposal

As to not stray too far from what already exists, our proposal consists in combining the aforementioned theories: Seq_{cvc5} , Seq_{z3} and Array_c , and applying our proposed changes. We outline our proposal in Figure 2.

6.3 Fragmenting the theory of sequences

As demonstrated in Wang and Appel's decision procedure [23], which was used to reason over arrays in C programs, and in the calculus developed by Sheng et al. [22], which was used to reason over vectors from smart contract verification. In some cases, a smaller fragment of the theory of sequences is sufficient to reason over arrays from many programming languages. Defining such a fragment is crucial, especially for solver developers, as it allows them to focus on developing reasoning capabilities for only the subset of the theory they require.

Given the considerable variation in operations supported by array-like data structures in programming languages, determining the precise set of symbols necessary and sufficient to represent such structures is challenging.

Building upon Array_c 's extension of the theory of arrays, which has been proven decidable when the verification condition has no index shifting⁶ [23], we can define a fragment of the theory of sequences based on our proposal, which can be reducible to the symbols of Array_c . The fragment is presented in Figure 3.

⁶Terms of the form: $\forall i : \text{Int}, 0 \leq i < \text{length}_S(s) - n \implies \text{nth}_S(i, s) = \text{nth}_S(i + n, s)$, where s is a sequence and n an integer literal

Symbol	Sort	Remark
seq.empty	Seq	
seq.const*	$\text{Int} \rightarrow \text{Elem} \rightarrow \text{Seq}$	replaces $\text{repeats}_S(\text{Array}_c)$
seq.unit	$\text{Elem} \rightarrow \text{Seq}$	$\text{seq.unit}(v) = \text{seq.const}(1, v)$
seq.len	$\text{Seq} \rightarrow \text{Int}$	
seq.get*	$\text{Seq} \rightarrow \text{Int} \rightarrow \text{Elem}$	replaces seq.nth (Seq_{cvc5} and Seq_{z3}) and $\text{nth}_S(\text{Array}_c)$
seq.set*	$\text{Seq} \rightarrow \text{Int} \rightarrow \text{Elem} \rightarrow \text{Seq}$	
seq.slice	$\text{Seq} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Seq}$	replaces seq.extract (Seq_{cvc5} and Seq_{z3}) and $\text{slice}_S(\text{Array}_c)$
seq.concat	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Seq}$	replaces seq.concat (Seq_{cvc5} and Seq_{z3}) and $\text{app}_S(\text{Array}_c)$
seq.at	$\text{Seq} \rightarrow \text{Int} \rightarrow \text{Seq}$	
seq.contains	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Bool}$	
seq.replace	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Seq}$	
seq.indexof	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Int}$	
seq.prefixof	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Bool}$	
seq.suffixof	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Bool}$	
seq.replace_all	$\text{Seq} \rightarrow \text{Seq} \rightarrow \text{Seq}$	Only present in Seq_{cvc5}
seq.rev	$\text{Seq} \rightarrow \text{Seq}$	Only present in Seq_{cvc5}
seq.update*	$\text{Seq} \rightarrow \text{Int} \rightarrow \text{Seq} \rightarrow \text{Seq}$	replaces seq.update (Seq_{cvc5})
seq.map*	$(\text{Elem}_1 \rightarrow \dots \rightarrow \text{Elem}_n \rightarrow \text{Elem}') \rightarrow \text{Seq}_1 \rightarrow \dots \rightarrow \text{Seq}_n \rightarrow \text{Seq}'$	replaces $\text{map}_f(\text{Array}_c)$, and seq.map (Seq_{z3})
seq.mapi*	$(\text{Int} \rightarrow \text{Elem}_1 \rightarrow \dots \rightarrow \text{Elem}_n \rightarrow \text{Elem}') \rightarrow \text{Int} \rightarrow \text{Seq}_1 \rightarrow \dots \rightarrow \text{Seq}_n \rightarrow \text{Seq}'$	replaces seq.mapi (Seq_{z3})
seq.fold_left	$(\alpha \rightarrow \text{Elem} \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{Seq} \rightarrow \alpha$	Only present (Seq_{z3})
seq.fold_lefti	$(\text{Int} \rightarrow \alpha \rightarrow \text{Elem} \rightarrow \alpha) \rightarrow \text{Int} \rightarrow \alpha \rightarrow \text{Seq} \rightarrow \alpha$	Only present (Seq_{z3})

Figure 2: Signature of the proposed Sequence theory. The * after a symbol means that the symbol's semantics are as described in the previous subsection. The sequence sorts $\text{Seq}_1 \dots \text{Seq}_n$ and Seq' have elements of the sorts $\text{Elem}_1 \dots \text{Elem}_n$ and Elem' respectively.

Other symbols can also be added, such as seq.mapi , which, as mentioned previously, can be reduced to map . Additionally, symbols like seq.fold_left , seq.fold_lefti , and seq.rev are common in array-like data structures, especially in functional programming languages. While such functions can always be defined as recursive functions, the potential impact of adding them to this fragment on decidability and reasoning efficiency needs to be further explored.

A proof of the soundness of the reductions of the symbols in the fragment is necessary, but we consider this to be beyond the scope of our paper. Nevertheless, this example can serve as a step towards defining a fragment of the theory of sequences tailored to represent array-like data structures.

Symbol	Reduction
<code>seq.empty</code>	<code>repeat(−, 0)</code>
<code>seq.const(l, v)</code>	<code>repeat(v, l)</code>
<code>seq.unit(v)</code>	<code>repeat(v, 1)</code>
<code>seq.len(s)</code>	<code>length(s)</code>
<code>seq.at(s, i)</code>	<code>ite(nth(s, i) = δ, repeat(−, 0), repeat(nth(s, i), 1))</code>
<code>seq.get(s, i)</code>	<code>nth(s, i)</code>
<code>seq.set(s, i, v)</code>	<code>update(i, s, v)</code>
<code>seq.slice(s, i, j)</code>	<code>slice(s, i, j)</code>
<code>seq.concat(s₁, s₂)</code>	<code>app(s₁, s₂)</code>
<code>seq.update(s₁, i, s₂)</code>	<code>app(slice(s₁, 0, i−1), app(s₂, slice(s₁, i + seq.len(s₂), seq.len(s₁) − 1)))</code>
<code>seq.map(f, s₁, ..., s_n)</code>	<code>map_f(s₁, ..., s_n)</code>

Figure 3: Fragment of the proposed theory of sequences and how its symbols can be reduced to those of `Arrayc`. δ represents a default value of the same sort as the return value of `nth`, which is returned when `nth` is applied outside its domain. `−` any value of the right sort.

7 Conclusion

In this paper, we have explored SMT theory design through the lens of the theory of sequences. We defined a set of criteria to consider when designing a theory and examined how partial functions are handled. Additionally, we proposed a variant of the theory of sequences based on these criteria, which we believe should be considered in the event of the theory’s standardization.

While the criteria we outlined should aid in theory design, they are not precise enough and are open to interpretation, particularly regarding notions such as user-friendliness. Ultimately, choices in theory design can be quite subjective.

References

- [1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. Conference Name: IEEE Std 754-2008.
- [2] Sergey Berezin, Clark Barrett, Igor Shikanian, Marsha Chechik, Arie Gurfinkel, and David L. Dill. A Practical Approach to Partial Functions in CVC Lite. *Electronic Notes in Theoretical Computer Science*, 125(3):13–23, July 2005.
- [3] N Børner, Vijay Ganesh, R Michel, and Margus Veanes. An SMT-LIB Format for Sequences and Regular Expressions. *Strings*, January 2012.
- [4] Bernard Boigelot, Pascal Fontaine, and Baptiste Vergain. Decidability of Difference Logic over the Reals with Uninterpreted Unary Predicates. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction – CADE 29*, Lecture Notes in Computer Science, pages 542–559, Cham, 2023. Springer Nature Switzerland.
- [5] Maria Paola Bonacina, Stephane Graham-Lengrand, and Natarajan Shankar. CDSAT for Nondisjoint Theories with Shared Predicates: Arrays With Abstract Length. *Satisfiability Modulo Theories workshop, CEUR Workshop Proceedings*, 3185, August 2022.
- [6] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What’s Decidable About Arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 427–442, Berlin, Heidelberg, 2006. Springer.

- [7] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 160–167, June 2015. ISSN: 1063-6889.
- [8] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, SMT '08/BPR '08*, pages 6–11, New York, NY, USA, July 2008. Association for Computing Machinery.
- [9] Guillaume Bury and François Bobot. Verifying models with dolmen. In Stéphane Graham-Lengrand and Mathias Preiner, editors, *Proceedings of the 21st International Workshop on Satisfiability Modulo Theories (SMT 2023) co-located with the 29th International Conference on Automated Deduction (CADE 2023), Rome, Italy, July, 5-6, 2023*, volume 3429 of *CEUR Workshop Proceedings*, pages 62–70. CEUR-WS.org, 2023.
- [10] Jürgen Christ and Jochen Hoenicke. Weakly Equivalent Arrays. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems*, Lecture Notes in Computer Science, pages 119–134, Cham, 2015. Springer International Publishing.
- [11] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*, pages 45–52, November 2009.
- [12] Stephan Falke, Carsten Sinz, and Florian Merz. A theory of arrays with set and copy operations. In Pascal Fontaine and Amit Goel, editors, *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories*, volume 20 of *EPiC Series in Computing*, pages 98–108. EasyChair, 2013.
- [13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 125–128, Berlin, Heidelberg, 2013. Springer.
- [14] Yeting Ge and Leonardo de Moura. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 306–320, Berlin, Heidelberg, 2009. Springer.
- [15] Silvio Ghilardi, Alessandro Gianola, Deepak Kapur, and Chiara Naso. Interpolation Results for Arrays with Length and MaxDiff. *ACM Transactions on Computational Logic*, 24(4):28:1–28:33, June 2023.
- [16] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Jan Van Leeuwen, editors, *Computer Science Today*, volume 1000, pages 366–373. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. Series Title: Lecture Notes in Computer Science.
- [17] Reiner Hähnle. Many-Valued Logic, Partiality, and Abstraction in Formal Specification Languages. *Logic Journal of the IGPL*, 13(4):415–433, July 2005. Conference Name: Logic Journal of the IGPL.
- [18] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning, LPAR'10*, pages 348–370, Berlin, Heidelberg, April 2010. Springer-Verlag.
- [19] John McCarthy. Towards a Mathematical Science of Computation. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, pages 21–28. North-Holland, 1962.
- [20] Philipp Rümmer and Thomas Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland, 2010*.
- [21] Peter H. Schmitt. A Computer-Assisted Proof of the Bellman-Ford Lemma, 2011. ISSN: 2190-4782.
- [22] Ying Sheng, Andres Nötzli, Andrew Reynolds, Yoni Zohar, David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Clark Barrett, and Cesare Tinelli. Reasoning About Vectors: Satisfiability Modulo a Theory of Sequences. *Journal of Automated Reasoning*, 67(3):32, September 2023.
- [23] Qinshi Wang and Andrew W. Appel. A Solver for Arrays with Concatenation. *Journal of Auto-*

mated Reasoning, 67(1):4, January 2023.