A theory of lists with combinators for SMT solvers

Submission for the POPL 2026 Student Research Competition

PIERRE GOUTAGNY*, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, France

1 INTRODUCTION

In order to solve a wide variety of problems, SMT solvers need theories to reason about complex data structures and operations. In particular, some implement efficient encodings of large collections of data, such as arrays with index-based operations [6], and strings with structural operations like concatenation or regular expressions [9]. This is, however, not the case for lists defined as recursive algebraic datatypes, equipped with the usual list operations found in functional programs. These operations can be written as compositions of fundamental combinators (`fold`, `map`, `filter`, etc.) that manipulate the content of lists without explicitly indexing their elements.

While state-of-the-art SMT solvers do provide ways to reason about lists, we argue that they suffer from significant limitations, and that they do not leverage the structure of compositional programs on lists to its full extent. In this work, we explore the strengths and limitations of current solvers capable of reasoning on collections of values, then introduce our first steps towards a new approach specifically tailored to lists with combinators, and fast unsatisfiability checking.

2 BACKGROUND AND RELATED WORK

Modern SMT solvers have several ways of representing lists, or list-like structures. However, the different approaches suffer from (sometimes prohibitive) limitations.

Algebraic datatypes and sequences. Solvers such as CVC5 or Z3 define a theory of *sequences* [1; 4; 5] built upon an underlying theory of algebraic datatypes. Lists are therefore built recursively with a `nil` constructor, and a `cons` or `concat` constructor. Most operations over lists can then be defined recursively, as in ML-like languages. This encoding enables writing constraints on lists of symbolic size, and constraints on the contents of these lists.

However, solving these can be inefficient with very large lists, especially in unsatisfiable instances, and lead to timeouts. Indeed, current SMT solvers do not implement inductive reasoning, which would help solve recursively defined constraints. Instead, they consider a candidate, then 'unroll' the definitions for it. Without a proper abstraction for list lengths, the solver may resolve to generating candidate lists of increasing lengths, and eventually time out after trying many list lengths.

String theory. Some solvers, such as Z3str4 [9], provide an efficient theory of strings, where strings are first-class elements, and not built recursively. However, these theories cannot be leveraged directly to get a theory of lists.

First, strings are made of letters that come from a finite alphabet, which is enough to represent ASCII or Unicode characters, but not the infinite set of integers.

Additionally, operations on strings, such as concatenation, slicing, regular expression matching, etc., usually rely on the structure of the string. The only operation on a string's *contents* in Z3's theory [3], for instance, is replacing a substring with another one. In particular, string theories do not define operations resembling a `fold`. However, string solvers like Z3str4 can efficiently handle strings of symbolic lengths with the help of a *Length Abstraction Solver*.

*ACM Student Member No. 2155597. Graduate student advised by Aymeric Fromherz and Raphaël Monat.

Author's address: Pierre Goutagny, Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000, Lille, France, pierre.goutagny@inria.fr.

Arrays. In the SMT theory of arrays, arrays are handled with a `store` and a `select` function [6][2, Fig. 3.3]. They can be indexed by any type, and have elements of any type, which essentially makes them functions. Because of this, arrays don't have an intrinsic length, and constraints must explicitly state to which subset of the domain they apply. On small examples, this approach seems to work, and even correctly answer UNSAT on some unsatisfiable constraints. However, since the model for an array is a function, retrieving all elements from a large array becomes inefficient.

Lists of concrete lengths. Finally, some solvers, such as Rosette [10], only allow lists of symbolic elements if they have a concrete length. With this approach, Rosette can lift Lisp-like concrete list operations (*e.g.* `cdr`, `car`) to lists of symbolic elements, and thus benefit from functions defined in the underlying Racket language. However, the user becomes responsible for finding suitable concrete lengths for symbolic lists.

3 OUR APPROACH

Our approach aims to combine strategies from string solvers, that reason well on lengths, with the ability of sequence solvers to handle integers and other data types. Using a theory of lists as abstract values with combinators, we describe a general method that alternates between finding suitable lengths, and finding a model for lists of these concrete lengths. To decrease solving time, we define rewriting-based optimizations that leverage the structure of list operation composition, and explore heuristics to reject unsatisfiable instances early.

3.1 Manipulating abstract lists with combinators

In our theory, lists are abstract values that are neither constructed from, nor breakable into, a head and tail. With this approach, the fundamental list operations are not `nil`, `cons`, and pattern matching, like in many ML-like languages. Rather, they are combinators that apply to whole lists, and are not defined recursively in the language: `fold`, `map`, `length`, `filter`.

Using this design choice, unrolling definitions is not always necessary. In particular, it allows us to reason on list lengths independently from their shape or content. It also becomes easier to recognize patterns and create optimizations that are specific to list operations, especially using rewriting. We show how to leverage these properties below.

3.2 Solver architecture

The main loop of the solver is composed of two phases: a Length Solver to find acceptable list lengths, and a Fixed-length List Solver to find lists of the given lengths.

Length Solver. In the first phase, the Length Solver starts by taking constraints over lists, and extracting arithmetic constraints over list lengths. For example, `length (map f l) = 2` is translated into $|l| = 2$, where $|l|$ is a symbolic integer variable representing `l`'s length. The Length Solver can then call an SMT solver to solve these constraints. If they are unsatisfiable, that is if no lengths can satisfy the length constraints, then no list can satisfy the list constraints. If they are satisfiable, the Length Solver gets a concrete model for possible list lengths.

Fixed-length List Solver. In the second phase, the Fixed-length List Solver takes as input the problem's constraints, and the length model described above. It encodes a list `l` of concrete length $\mathbf{n}$ as $\mathbf{n}$ SMT variables $l_1, \ldots, l_\mathbf{n}$. The list constraints are then encoded in the SMT solver as follows:

- `length l` is encoded as the SMT literal representing concrete length $\mathbf{n}$.
- `map f l` is encoded as the conjunction of assertions mapping each $f(l_i)$ to a new variable $l_\mathbf{i}^{\text{map}}$. For example, when encountering (`map (fun x -> x+1) l`), and knowing $|l| \mapsto \mathbf{3}$ in the length model, the solver generates $\left(l_\mathbf{1}^{\text{map}} = l_1 + \mathbf{1}\right) \wedge \left(l_\mathbf{2}^{\text{map}} = l_2 + \mathbf{1}\right) \wedge \left(l_\mathbf{3}^{\text{map}} = l_3 + \mathbf{1}\right)$.

- `fold f i l` is encoded as the 'unrolling' of the `fold` operation. For example, (`fold (+) 0 l`), where $|l| \mapsto \mathbf{3}$ in the length model, is encoded as $(((0 + l_1) + l_2) + l_3)$.

After this encoding step, the Fixed-length List Solver calls an SMT solver with the generated constraints. If the constraints are satisfiable and the SMT solver returns a model with values for every element of `l`, the list solver can stop and return a model for `l` using these elements. If the constraints are unsatisfiable, then the Length Solver must generate a new length model, if one is available, while possibly factoring in the Fixed-length List Solver's negative result.

Length bounds and timeouts. If, at some point, every satisfying length model has been tried, and none leads to a satisfying model for list elements, then the list solver can return UNSAT. Otherwise, it may run indefinitely, or be stopped by a timeout and answer UNKNOWN.

3.3 Optimizations and heuristics

The method described above is designed to work in the general case, and will always eventually return SAT, UNSAT, or UNKNOWN if a timeout is set. However, this is not always the most efficient method, especially for large or unsatisfiable problems. In this section, we propose optimizations and heuristics for these cases.

Composition rules. Using the fact that our language has a limited set of pure list operations, it is possible to write general rewriting rules to reduce solving time. For instance, it is possible to rewrite terms of the shape `map f (map g l)` as `map (fun x -> f (g x)) l`. For a list l of length n, using the Fixed-length List Solver on the left-hand side will create n constraints to encode `map g`, then n more to encode `map f`. On the right-hand side, the same approach only needs n constraints.

Similar rewritings are possible for compositions of `fold`, `map`, `length`, and `filter`. We plan to study the choice of the sub-expressions to rewrite and the order of the rewritings, as well as their impact on performance.

Unsatisfiable problems. The composition simplifications described above are designed to apply in many cases, and reduce the overall solving time. However, if list lengths are bounded, the domain of the Length Solver grows exponentially with the number of lists, and if they are unbounded, the solver may never terminate. For instance, if a problem contains N lists of length smaller than B, there are B^N possible length models. In the worst case, if no length model allows for a satisfiable Fixed-length problem, the Fixed-length List Solver will be called B^N times before returning UNSAT. We are therefore interested in heuristics that will detect unsatisfiable problems as early as possible through state-space reduction.

4 EXPERIMENTAL EVALUATION AND USE CASE

To verify our approach empirically, we are developing an implementation of this ongoing work in OCaml, relying on Z3 [5] as our SMT solver backend. Using both purpose-made and real-world programs that manipulate lists, we aim to perform an experimental evaluation of the general approach described above, compare it with state-of-the-art solvers, and identify the practical impact of each heuristics.

Lastly, our goal is eventually to integrate this solver in CUTECat [7], our concolic testing engine for implementations of computational laws. Implementing fiscal or social benefits law, for instance, can require reasoning on lists of individuals, periods of employment, tax brackets, etc. Using our list solver would therefore enable the verification of real-world programs that implement these laws in the Catala domain-specific language [8].

REFERENCES

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. Cvc5: A Versatile and Industrial-Strength SMT Solver. In *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* (2022) *(Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7*. Technical Report. Department of Computer Science, The University of Iowa. https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-07-07.pdf

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *SMT-LIB website — Unicode Strings*. https://smt-lib.org/theories-UnicodeStrings.shtml

[4] Nikolaj Bjørner, Vijay Ganesh, Raphael Michel, and Margus Veanes. 2012. An SMT-LIB Format for Sequences and Regular Expressions. (2012).

[5] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. https://doi.org/10.1007/978-3-540-78800-3_24

[6] Leonardo de Moura and Nikolaj Bjørner. 2009. Generalized, Efficient Array Decision Procedures. In *2009 Formal Methods in Computer-Aided Design* (2009-11). 45–52. https://doi.org/10.1109/FMCAD.2009.5351142

[7] Pierre Goutagny, Aymeric Fromherz, and Raphaël Monat. 2025. CUTECat: Concolic Execution for Computational Law. In *Programming Languages and Systems* (Cham, 2025), Viktor Vafeiadis (Ed.). Springer Nature Switzerland, 31–61. https://doi.org/10.1007/978-3-031-91121-7_2

[8] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021. Catala: A Programming Language for the Law. 5 (2021), 77:1–77:29. Issue ICFP. https://doi.org/10.1145/3473582

[9] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed String Solver. In *Formal Methods* (Cham, 2021), Marieke Huisman, Corina P˘ asăreanu, and Naijun Zhan (Eds.). Springer International Publishing, 389–406. https://doi.org/10.1007/978-3-030-90870-6_21

[10] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014-06-09). 530–541. https://doi.org/10.1145/2594291.2594340