Polyregular Model Checking

Aliaume Lopez(✉) and Rafał Stefański

University of Warsaw, Warsaw, Poland
ad.lopez@uw.edu.pl

Abstract. We introduce a high-level language with Python-like syntax for string-to-string, polyregular, first-order definable transductions. This language features function calls, boolean variables, and nested for-loops. We devise and implement a complete decision procedure for the verification of such programs against a first-order specification. The decision procedure reduces the verification problem to the decidable first-order theory of finite words (extensively studied in automata theory), which we discharge using either complete tools specific to this theory (MONA), or to general-purpose SMT solvers (Z3, CVC5).

1 Introduction

String manipulating programs of low complexity are ubiquitous in modern software. They are often used to transform data and do not perform complex computations themselves. In this paper, we are interested in verifying Hoare triples for such string manipulating programs, i.e. specifications of the form $\{P\}$ `code` $\{Q\}$, where P and Q are pre- and post-conditions, meaning that whenever the input satisfies property P, the output of the program satisfies property Q.

Regularity Preserving Programs. One particularly interesting class of specifications in the case of string-to-string functions are *regular languages*, which can be efficiently verified using automata-based techniques. We say that a function f is *regularity preserving* if it preserves regular languages under pre-image, i.e. if $f^{-1}(L)$ is regular for all regular languages L. For regularity preserving functions, the verification of a Hoare triple $\{L_P\}$ f $\{L_Q\}$ can be reduced to the nonemptiness problem of the language $L_P \cap f^{-1}(L_Q)$, where L_P and L_Q are regular languages. This is a well-studied problem in the literature, and is at the core of several more involved techniques [2,13,18]. The key challenge of this approach is that there exist uncomputable regularity preserving functions, so such approaches will only work on classes of functions for which pre-images of regular languages are (relatively) efficiently computable. Usually, these classes come from generalisation of automata models to functions, also known as *string-to-string transducers*.

A. Lopez—Supported by the Polish National Science Centre (NCN) grant "Polynomial finite state computation" (2022/46/A/ST6/00072).
R. Stefański—Supported by the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666).
This document uses knowledge: notion points to its *definition*.

© The Author(s) 2025
R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15933, pp. 3–25, 2025.
https://doi.org/10.1007/978-3-031-98682-6_1

String-to-String Transducer Models. There is a wide variety of models for string-to-string transducers [24], and one of the most prominent ones is called *linear regular functions*, that are equivalently defined using two-way finite transducers (2DFTs) [26], streaming-string-transducers (SSTs) [1], or linear regular list functions [9]. Notably, Alur and Černý have proven that SSTs have a PSPACE-complete model checking problem when the functions are given as SSTs, and the specifications are given as automata [2, Theorem 13]. This was used for instance by Chen, Taolue, Hague, Lin, Rümer, and Wu to study *path feasibility* in string-manipulating programs [13].

A similar approach was used by Jeż, Lin, Markgraf, and Rümmer, who leveraged the *rational functions* (a strict subclass of linear regular functions) to study programs manipulating strings with infinite alphabets [19]. Remark that in the setting of infinite alphabets, the landscape of automata and transducers is much more complex: In partcicular, the class of languages recognised by two-way automata is stronger that the class of languages recognised by one-way automata, and has undecidable emptiness [8, Figure 1.1].

One limitation of the linear regular functions is that they only allow for linear growth of the output, excluding many useful string-manipulating programs. The class of *polyregular functions* is an interesting generalisation of linear regular functions that allows for polynomial behaviour, and is much closer to real life string manipulating programs. The model is relatively old, first introduced in [16], and has recently gained a lot of traction now that several other characterizations have been obtained [6,7].[1] However, the proof of the regularity preserving property for polyregular functions is of theoretical nature (no implementation or complexity bounds are given), and writing programs using any of the existing equivalent definitions of polyregular functions is cumbersome and error-prone. Because polyregular functions can succinctly encode formulas in first-order logic on words, and since the satisfiability problem for such formulas is known to be TOWER-complete [27, Theorem 13.5], one can expect that verifying polyregular functions to be quite complex.

MSO vs FO. Instead of using the full power of regular languages (defined equivalently using finite automata, monadic second order logic (MSO), finite monoid recognition, or regular expressions [11,21,28,30]), we will use specifications written in first-order logic (FO) on finite words. A cornerstone result of the theory is establishing the equivalence between languages described in this logic, *star-free languages*, and *counter free automata* [22,25,29]. The advantage of using this weaker specification model is twofold: first, it allows us to focus on a simpler class of star-free polyregular functions[2], which are easier to work with in practice. Second, it allows us to reduce the satisfiability of a Hoare triple to

[1] Note that for this extended model, being regularity preserving is tightly connected to being closed under function composition [17, Proposition III.3], and this closure under composition was one of the surprising conclusions of [7].

[2] The notion of being *star-free* has been extended to various classes of transducers, see [7,9,12,24].

the satisfiability of a *first-order* formula on finite words, for which one can use general purpose SMT solvers, in addition to automata based tools (MONA) which also work for the MSO logic on words. Even though the SMT solver are *incomplete*, they can, in some cases, lead to faster decision procedures. Indeed, the satisfiability problem for first-order logic on finite words, while decidable, is TOWER-complete [27, Theorem 13.5].

Contributions. In this paper, we introduce a high-level programming language for implementing star-free polyregular functions in a Python-like syntax, including features such as boolean variables, index variables, immutable list variables, function calls, and nested for-loops. The language was carefully designed not become too expressive – this is ensured by a number of syntactic restrictions and a novel type system for index variables. We show that this language can be compiled into one of the equivalent definitions of polyregular functions (namely, simple for-programs), which does not allow for function calls nor list variables. We also provide an implementation of the previously known abstract result stating that polyregular functions are regularity preserving (in the case of star-free functions and languages), being careful about the complexity of the transformations. Finally, we reduce the verification of Hoare triples to the satisfiability of first-order formulas on words. Since we are using first-order logic as a target language, we are not restricted to using automata based tools like *MONA* [20], but can also employ general purpose SMT solvers like *Z3* [23] and *CVC5* [32], generating proof obligations in the `SMT-LIB` format [3].

All the steps described above have been implemented in a `Haskell` program, and tested on a number of examples with encouraging results.[3] While this is not a tool paper, we believe that the proof-of-concept implementation is a good starting point to demonstrate the viability of our approach, and we believe that there is a potential for further investigations in this direction.

Outline. The structue of the paper is as follows. We introduce our high-level language in Sect. 2. In Sect. 3, we recall the theory of polyregular functions by introducing them in terms of simple for-programs and FO-interpretations. We will also provide an efficient reduction of the verification of Hoare triples to the satisfiability of a first-order formula on words in Sect. 3.4. In order to verify for-programs, we compile them into simple for-programs in Sect. 4, and then compile simple for-programs into FO-interpretations in Sect. 5. Then, in Sect. 6, we present tests of our implementation on various examples, discussing the complexity of the transformations and the main bottlenecks of our approach. Finally, in Sect. 7, we discuss potential optimizations and future work.

[3] An anonymized version of our code is available at https://github.com/AliaumeL/polyregular-model-checking.

2 High Level For Programs

In this section, we introduce our high-level language for describing list-manipulating functions which can be seen as a subset of `Python`, which we call *(high-level) for-programs*. Our goal is to reason algorithmically about the programs written in this language, so it needs to be highly restricted. To illustrate those restrictions, let us present in Fig. 1 a comprehensive example written in a subset of `Python`.[4]

```python
def getBetween(l, i, j):
    """ Get elements between i and j """
    for (k, c) in enumerate(l):
        if i <= k and k <= j: ①
            yield c ②

def containsAB(w):
    """ Contains "ab" as a subsequence """
    seen_a = False ③
    for (x, c) in enumerate(w):
        if c == "a": ④
                seen_a = True ⑤
        elif seen_a and c == "b":
            return True
    return False

def subwordsWithAB(word):
    """ Get subwords that contain "ab" """
    for (i,c) in enumerate(word): ⑥
        for (j,d) in reversed(enumerate(word)): ⑦
            s = getBetween(word, i, j) ⑧
            if containsAB(s):
                yield s
```

Fig. 1. A small Python program that outputs all subwords of a given word containing `ab` as a scattered subword

.

For the sake of readability, we implicitly coerce generators (created using the `yield` keyword) to lists. Our programs will only deal with three kinds of values: booleans ($\mathbb{B}$), non-negative integers ($\mathbb{N}$), and *(nested) words* ($\mathcal{W}$), i.e. characters ($\mathcal{W}0$), words ($\mathcal{W}_1$), lists of words ($\mathcal{W}_2$), etc. These lists can be created by *yielding*

[4] The corresponding program in the syntax accepted by our solver is given in the full version of this paper.

values in a loop, such as in ② of Fig. 1. In order to ensure decidable verification of Hoare triples,[5] we also will enforce the following conditions, which are satisfied in our example:

(I) **Loop Constructions.** We only allow `for` loops iterating forward or backward over a list, as in ⑥ and ⑦. In particular, `while` loops and recursive functions are forbidden, which guarantees termination of our programs.

(II) **Mutable Variables.** The only mutable variables are booleans. The values of integer variables are introduced by the `for` loop as in ⑥, and their values are fixed during each iteration. Mutable integer variables could serve as unrestricted counters, resulting in undecidable verification. Similarly, we prohibit mutable list variables, as their lengths could be used as counters. However, we still allow the use of immutable list variables, as in ⑧.

(III) **Equality Checks.** We disallow equality checks between two nested words, unless one of them is a constant expression. This is what happens in point ④ of our Fig. 1. Without this restriction, verification would also be undecidable. More generally, classical string *algorithms* (edit distance, string matching, longest common subsequence, etc.) should not be expressible in our language, since one can easily derive an equality check from them.

(IV) **Integer Comparisons.** The only allowed operations on integers are usual comparisons operators (equality, inequalities). However, we only allow comparisons between integers that are indices of the same list. Every integer is associated to a list expression. For instance, in points ⑥ and ⑦ of our example, the variables i and j are associated to the same list variable `word`. Similarly, for the comparison of point ① to be valid, the variables k, i, and j should all be associated to the same list variable l.

To ensure this compatibility, we designed the following type system, containing Booleans, nested words of a given depth (characters are of depth 0), and integers associated to a list expression (the set of which is denoted by LExpr):

$$\tau ::= \mathsf{Bool} \mid \mathsf{Pos}_o \mid \mathsf{List}_n \quad n \in \mathbb{N}, o \in \mathsf{LExpr} \quad .$$

These types can be inferred from the context, except in the case of function arguments, in which case we explicitly specify to which list argument integer variables are associated. Without this restriction, the equality predicate between two lists can be redefined.

(V) **Variable Shadowing.** We disallow shadowing of variable names, as it could be used to forge the origin of integers, leading to unrestricted comparisons .

(VI) **Boolean Arguments.** We disallow functions to take boolean arguments, as it would allow to forge the origin of integers, by considering the function `switch(b, l1, l2)` which returns either `l1` or `l2` depending on the value of `b` .

[5] Using first-order logic on words as a specification language.

(VII) **Boolean Updates.** Boolean variables are initialized to `false` as in ③, and once they are set to `true` as in ⑤, they cannot be reset to `false`. We depart here from the semantics of Python by considering lexical scoping of variables; in particular a variable declared in a loop is not accessible outside this loop.
This restriction allows us to reduce the verification problem to the satisfiability of a first-order formula on finite words. This problem is not only decidable but also solvable by well-engineered existing tools, such as automata-based solvers (e.g., MONA) and classical SMT solvers (e.g., Z3, and CVC5). Without this restriction, the problem would require the use the monadic second order logic on words which is still decidable but not supported by the SMT solvers.

Formal Syntax and Typing. We extend the typing system to functions by grouping input positions with the list they are associated to. For instance, the function `getBetweenIndicesBeforeStop(l, i, j)` has type $(\mathsf{List}_2, 2) \to \mathsf{List}_2$, that is, we are given an input list l together with two pointers to indices of this list. Similarly, the function `containsAB(w)` has type $(\mathsf{List}_1, 0) \to \mathsf{Bool}$, while the function `subwordsWithAB(word)` has type $(\mathsf{List}_1, 0) \to \mathsf{List}_2$. We implemented a linear-time algorithm for the type checking and inference problems.

The formal syntax of our language is given in the full version of this paper. They define the syntax of *boolean expressions* (BExpr), *constant expressions* (CExpr), *list expressions* (LExpr), and *control statements* (Stmt). For readability, we distinguish boolean variables $\mathbb{V}_{\mathrm{bool}}$ $(b, p, q, \dots)$, position variables $\mathbb{V}_{\mathrm{pos}}$ $(i, j, \dots)$, list variables $\mathbb{V}_{\mathrm{list}}$ $(x, y, u, v, w, \dots)$, and function variables $\mathbb{V}_{\mathrm{fun}}$ $(f, g, h, \dots)$. A *for-program* is a list of function definitions together with a *main* function of type $\mathsf{List}_1 \to \mathsf{List}_1$.

Semantics. Given an *evaluation context* that assigns values (functions, positions, booleans, nested words) to variables, the semantics of boolean expressions into booleans $\mathbb{B}$, constant expressions into nested words, and list expressions into nested words pose no difficulty. For the control statements, there is a crucial design choice regarding the semantics of backward iteration. While the semantics of forward iteration is unambiguous, the backward iteration $\mathsf{for}^{\leftarrow}\ (i, x)\ \mathsf{in}\ l\ \mathsf{do}\ s$ could be understood in two different ways, provided that l evaluates to a list $[x_0, \dots, x_k]$:

- Executing the statement s for pairs $(k, x_k), \dots, (0, x_0)$. This corresponds to Python's `for (i,x) in reversed(enumerate(l))` ;
- Executing the statement s for pairs $(0, x_k), \dots, (k, x_0)$. This corresponds to Python's `for (i,x) in enumerate(reversed(l))`.

As shown in our example program, we use the first interpretation (see ⑦). In fact, the second interpretation would allow us to define the equality predicate between two lists, leading to undecidable verification.

3 Polyregular Functions

To obtain a decision procedure for the verification of Hoare triples for for-programs, we will prove that they can be compiled to *first-order polyregular functions*– a class of transductions introduced in [7] whose model checking problem is decidable [7, Theorem 1.7]. We provide two equivalent definitions of the first-order polyregular functions: one using first-order simple for-programs [7, p. 19] and one using the logical model of first-order string-to-string interpretations [6, Definition 4], the equivalence of which was proven in [7].

To make the models more suitable for large alphabets (such as the Unicode characters), we present them in a symbolic setting (which uses a simplified version of the ideas presented in [15] or in [5, Section 3.1]). This will dramatically reduce the size of the first-order string-to-string interpretations, and in turn, of the first-order formula that we will feed to the solvers. We will prove in Sect. 5 that every first-order simple for-programs can be transformed into a first-order string-to-string interpretation in the symbolic setting. We believe that the other inclusion should also hold, but do not prove it, as it is out of this paper's scope.

3.1 Symbolic Transductions

Consider the program in Fig. 2, which swaps all as to bs in a string. Even though it operates on the entire Unicode alphabet, it only distinguishes between three types of characters: `a`, `b` and the rest. To formalize this observation, we model the Unicode alphabet as an infinite set $\mathcal{D}$, and we define a function $T : \mathcal{D}^* \to \mathcal{D}^*$ to be *supported by* a set $A \subseteq \mathcal{D}$, if for every function $f : \mathcal{D} \to \mathcal{D}$ that does not *touch* elements of A (i.e. $\forall_{a \in A}, f^{-1}(a) = \{a\}$), it holds that:

$$\forall_w \quad T(f^*(w)) = f^*(T(w)) \quad ,$$

where f^* is the extension of f to $\mathcal{D}^*$, defined by applying f to every letter.

Functions defined by for-programs (of type $\mathsf{List}_1 \to \mathsf{List}_1$) are supported by the finite set A of letter constants that they use. This is also going to be the case for the simple for-programs that we introduce in Sect. 3.2. In Sect. 3.3, we will define a version of the first-order string-to-string interpretations in a way that only depends on the size of their support A, and not on the number of the Unicode characters.

```
def asToBs(w):
    for (i, c) in enumerate(w):
        if c == 'a':
            yield 'b'
        else:
            yield c
```

Fig. 2. The `swapAsToBs` program.

3.2 First-Order Simple For-Programs

First-order simple for-programs—originally introduced in [7, p. 19]—can be seen as simplified[6] version of the for-programs. The main difference is that the simple for-programs only define transductions of type $\mathsf{List}_1 \to \mathsf{List}_1$. Here is an example in a Python syntax:

```
# The program reverses all space-separated words
# in the input string. e.g
#        "hello world" -> "olleh dlrow"
seen_space_top = False ①
# first we handle all words except of the final one
for i in input: ②
    seen_space = False ③
    if label(i) == ' ': ④
        for j in reversed(input): ⑤
            if j < i:
                if label(j) == ' ':
                    seen_space = True
                if not seen_space:
                    print(label(j)) ⑥
        print(' ') ⑦

# then we handle the final word
for j in reversed(input):
    if label(j) == ' ':
        seen_space_top = True
    if not seen_space_top:
        print(label(j))
```

We disallow constructing intermediate word-values, there are no variables of type List_n for any n, and it is not possible to define functions (other than the main function). As a consequence, the for-loops can only iterate over the positions of the input word as in ① and ②. The character at a given position can be accessed using the keyword label, whether when testing it (⑤) or when printing it in (⑥). As we are considering a restriction of for-programs, we only allow comparing labels to constant characters (R. III). Finally, we only allow introducing boolean variables at the top of the program (①) or at the beginning of a for loop (③).

3.3 First-Order String-To-String Transductions

First-order string-to-string interpretations forms an other model that defines functions $\mathcal{D}^* \to \mathcal{D}^*$. It is based on the *first-order logic on words* (FO), the

[6] Actually, the for-programs were designed as an extended version first-order simple for-programs.

syntax of which we recall in Fig. 3. To evaluate such a formula φ on a word $w \in \mathcal{D}^*$ we perform the quantifications over the positions in w. The predicates $x = y$ and $x < y$ have the natural meaning, and $x =_L \mathsf{a}$ is checks if the x-th letter of w is equal to a. Let us recall that the *quantifier rank* of a formula is the maximal number of nested quantifications in it.

$$\begin{aligned} \varphi,\ \psi := &\ \forall_x\ \varphi \mid \exists_x\ \varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \\ &\mid x = y \mid x < y \mid x =_L \mathsf{a}, \text{ where } \mathsf{a} \in \mathcal{D} \end{aligned}$$

Fig. 3. First-order logic on words.

An important property of FO, is that it has decidable *emptiness*, i.e. given a formula φ, one can decide if there is a word w such that φ holds for w. For finite alphabets, this property is well-know [11], and for the infinite alphabet $\mathcal{D}$ it is the consequence of the finite-alphabet case.

Having discussed the first-order logic on words, we are now ready to define the first-order string-to-string interpretations.

Definition 1. *A* first-order string-to-string interpretation *consists of:*

1. *A finite set of* character constants $A \subset_{fin} \mathcal{D}$.
2. *A finite set* T *of* tags.
3. *An* arity function $\mathsf{ar} : T \to \mathbb{N}$.
4. *An* output function $\mathsf{out} : T \to A + \{1, \dots, \mathsf{ar}(t)\}$.
5. *A* domain formula $\varphi^t_{\mathsf{dom}}(x_1, \dots, x_{\mathsf{ar}(t)})$ *for every tag* $t \in T$.
6. *An* order formula $\varphi^{t,t'}_{\leq}(x_1, \dots, x_{\mathsf{ar}(t)}, y_1, \dots, y_{\mathsf{ar}(t')})$ *for every* $t, t' \in T$.

The order and domain formulas should only use constants from A.

The interpretation's output for a word $w \in \mathcal{D}^*$ is obtained as follows:

1. Take the set $P = \{1, \dots, |w|\}$ of the positions in w, and construct the set of *elements* as the set $T(P) = (t : T) \times P^{\mathsf{ar}(t)}$ of all tags from T equipped with position tuples of the appropriate arity.
2. Filter out the elements that do not satisfy the domain formula.
3. Sort the remaining elements according to the order formula. Typically, we want the order formula to define a total order on the remaining elements of $T(P)$ – if this is not the case, the interpretation returns an empty word.
4. Assign a letter to each element according to the output function: For an element $t(p_1, \dots, p_k)$, we look at of $\mathrm{out}(t)$: If it returns $a \in A$ the output letter is a. If it returns $i \in \{1, \dots, k\}$, we copy the output letter from the p_i-th position of the input.

For example, let us present a first-order word-to-word interpretation for the function swapAsToBs in Fig. 4. It has two tags printB and copy, both of arity 1. The element printB(x) outputs the letter b and copy(x) outputs the letter of x-th position of the input word. The element printB(x) is present in the output if x is labelled with the letter b in the input, otherwise the element copy(x) is present: The tags are sorted by their positions, with ties resolved in favour of printB.

$$\mathrm{out}(\texttt{printB}) = \texttt{b} \quad \mathrm{out}(\texttt{copy}) = 1$$

$$\varphi^{\texttt{printB}}_{\mathrm{dom}}(x) : x =_L \texttt{b} \quad \varphi^{\texttt{copy}}_{\mathrm{dom}}(x) : x \neq_L \texttt{b}$$

$\varphi_{\leq}$	printB(x_1)	copy(x_1)
printB(x_2)	$x_1 \leq x_2$	$x_1 < x_2$
copy(x_2)	$x_1 \leq x_2$	$x_1 \leq x_2$

Fig. 4. The swapAsToBs interpretation.

3.4 Hoare Triple Verification

We say that the Hoare triple $\{\varphi\}\ \ F\ \ \{\psi\}$ is valid if for every word w that satisfies φ, the output $F(w)$ satisfies ψ. An important property first-order string-to-string interpretations is that they admit a direct reduction of the *first-order Hoare triple* verification problem to the emptiness problem for the first-order logic on words [7, Theorem 1.7]. However, the resulting construction is not efficient. We provide a direct construction of a first-order formula $\chi(\phi, F, \psi)$ that is unsatisfiable if and only if the triple $\{\phi\}\ \ F\ \ \{\psi\}$ is valid. Moreover, the size and the quantifier rank of χ are bounded by the following low-degree polynomials:

$$\mathsf{qr}(\chi) \leq \max\left(\mathsf{qr}(\phi),\ \mathsf{qr}(\psi) \cdot (\mathsf{ar}(F) + 1) + \mathsf{qr}(F)\right) \quad |\chi| = \mathcal{O}(|\phi| + |F| \cdot |\psi|)$$

Here $|F|$ denotes the sum of the sizes of formulas in F, $\mathsf{qr}(F)$ denotes quantifier depth of the deepest formula in F, and $\mathsf{ar}(F)$ denotes the maximal arity of the tags in F.

To construct the formula χ, we introduce a *pullback operator* $\pi(F, \psi)$ that transforms the formula ψ applied to the output F, to a formula $\pi(F, \psi)$ that can be applied directly the input word, corresponding to a form of *weakest precondition* [31, Chapter 7]. The pull-back operation is defined in such a way that $F(w)$ satisfies ψ if and only if w satisfies $\pi(F, \psi)$. Once we have the pull-back operation, we can define $\chi(\phi, F, \psi)$ as $\phi \wedge \neg\pi(F, \psi)$. In the rest of this section, we show how to efficiently construct $\pi(F, \psi)$.

Naïve Pullback Definition. Let us start with a simple but inefficient construction of the pullback operation. Every position from $F(w)$ corresponds to a tag t and a tuple of $\mathsf{ar}(t)$ positions of the input word w, so we can replace each quantification in ψ with a conjunction or disjunction over the tags, and use respectively the order formula and output function to implement the predicates over positions of $F(w)$. For example:

$$\forall_x, \psi \quad \rightsquigarrow \bigwedge_{t_x \in T} \forall_{x_1, \ldots, x_{\mathsf{ar}(t)}} \left(\varphi^t_{\mathsf{dom}}(x_1, \ldots, x_{\mathsf{ar}(t)}) \Rightarrow \psi'_t\right)$$

A similar transformation can be done for the existential quantifications. Then, one can implement the $\leq$ predicate by consulting the order formula:

$$x \leq y \quad \rightsquigarrow \psi_{\leq}^{t_x,t_y}(x_1,\ldots,x_{\mathsf{ar}(t_x)},y_1,\ldots,y_{\mathsf{ar}(t_y)})$$

Similarly the $=_L$ predicate can be handled by consulting the output function, and $x = y$ predicate can be handled by comparing equality of the tags and the positions of x and y. This construction, although correct, is unfortunately inefficient: Replacing each quantification with a disjunction or conjunction over tags, results in an exponential blow-up of the formula.

Efficient Pullback Definition. Let us introduce an additional finite sort T to the logic, which allows us to quantify over the tags using $\forall_{t\in T}\varphi$ and $\exists_{t\in T}\varphi$. This does not add expressive power to the logic, as the new quantifiers can be replaced by a finite conjunction (resp. disjunction) that goes through the tags. However, this new sort will allow us to construct the pullback operator in a more efficient way, that can be understood by the solvers (we discuss it in more details at the end of this section). With the new sort of tags, we can pull back the quantifiers in the following way:

$$\pi(F,\forall_x\psi) \quad = \forall_{t_x\in T}\,\forall_{x_1,\ldots,x_{\mathsf{ar}(F)}}\left(\mathsf{dom}(t_x,x_1,\ldots,x_{\mathsf{ar}(F)})\Rightarrow\pi(F,\psi)\right)$$

where dom is the following predicate based on the domain formula:

$$\mathsf{dom}(t,x_1,\ldots,x_{\mathsf{ar}(t)}) := \bigvee_{t'\in T}\left(t=t'\wedge\varphi_{\mathsf{dom}}^{t'}(x_1,\ldots,x_{\mathsf{ar}(t')})\right)$$

In order to implement the atomic predicates, we use formulas similar to dom, but based the order formula and output function:

$$\pi(F,x\leq y) = \bigvee_{t_1,t_2\in T}\left(t_x=t_1\wedge t_y=t_2\wedge\varphi_{\leq}^{t_1,t_2}(x_1,\ldots,x_{\mathsf{ar}(t_1)},y_1,\ldots,y_{\mathsf{ar}(t_2)})\right)$$

$$\pi(F,x=_L\mathsf{a}) = \left(\bigvee_{t\in T\wedge\mathsf{out}(t)=\mathsf{a}} t=t_x\right)\vee\left(\bigvee_{t\in T\wedge\mathsf{out}(t)\notin A}(t=t_x\wedge x_{\mathsf{out}(t)}=_L\mathsf{a})\right)$$

This way, we push the disjunction over tags all the way down in the formula, thus avoiding the exponential blow-up of the naïve approach.

Encoding Tags. Finally, let us briefly discuss how we handle the tags in the formulas fed to solvers. For the SMT-solvers, we use the `smtlib v2.6` format with logic set to `UFDTLIA` [3], which allows us to add finite sorts and quantify over them. For the MONA solver, which only supports the sort of positions, we encode the tags as the first $|T|$ positions of the input word. The pertinence of this choice of encoding will be discussed in Sect. 6.

4 From High Level to Low Level For Programs

In this section, we provide a compilation from high-level for-programs to simple for-programs. To smoothen the conversion, we introduce *generator expressions* to the language, as a way to inline function calls. We distinguish between nested-word generators $\langle s \rangle_{\mathrm{l}}$ and boolean generators $\langle s \rangle_{\mathrm{b}}$.

Generator Expressions. Let us briefly discuss the new typing rules and semantics of these generator expressions. The meaning of $\langle s \rangle_{\mathrm{l}}$ is to evaluate the statement s in the current context and collect its output. For instance, $\langle \mathsf{return}\ x \rangle_{\mathrm{l}}$ is equivalent to x, and $\langle \mathsf{yield}\ x \,;\, \mathsf{yield}\ y \rangle_{\mathrm{l}}$ is equivalent to $\mathsf{list}(x, y)$. Similarly, $\langle s \rangle_{\mathrm{b}}$ is used to evaluate a boolean statement and return its value. The type of a generator expression is equal to the type of the statement s it contains. Importantly, when evaluating the statement s in a generator, we hide all boolean variables from the evaluation context. In particular, $\mathsf{let\ mut}\ b = \mathsf{false\ in\ return}\ \langle \mathsf{return}\ b \rangle_{\mathrm{b}}$ is an *invalid program*, because the variable b is undefined in the context of the generator expression $\langle b \rangle_{\mathrm{b}}$. The formal typing rules of generator expressions can be found in the full version of this paper.

Hiding the booleans from the context, ensures that the evaluation order of the expressions is irrelevant, allowing us to freely substitute expressions during the compilation process.

Rewriting Steps. We will convert for-programs to simple for-programs by a series of rewriting steps listed below. While most of the steps make can be applied to any for-program, some of them only apply to programs of type $\mathsf{List}_1 \to \mathsf{List}_1$.

(A) *Elimination of Literal Equalities*, i.e., of expressions $c =_{\mathrm{lit}} o$ where $c \in \mathsf{CExpr}$ and $o \in \mathsf{LExpr}$. This is done by replacing those tests with a call to a function that checks for equality with the constant c by traversing its input. We define these functions by induction on c. Note that this is only possible because equalities are always between a variable and a constant (R. III).

(B) *Elimination of Literal Productions*, i.e., of constant expressions in the construction of LExpr, except single characters. This is done by replacing a constant c by a function call. For instance, $\mathsf{list}(\mathsf{char}(a_1), \mathsf{char}(a_2))$ is replaced by a call to a function with body $\mathsf{yield}\ \mathsf{char}(a_1) \,;\, \mathsf{yield}\ \mathsf{char}(a_2)$.

(C) *Elimination of Function Calls*, by replacing them with generator expressions. Given a function f with body s and arguments $x_1, \ldots, x_n$, we replace a call $f(a_1, \ldots, a_n)$ by $\langle s[a_1/x_1, \ldots, a_n/x_n] \rangle_{\mathrm{l}}$ (or $\langle \cdots \rangle_{\mathrm{b}}$ for boolean functions). This is valid because functions do not take booleans as arguments (R. VI).

(D) *Elimination of Boolean Generators*. Note that $\langle s \rangle_{\mathrm{b}}$ can only appear in a conditional test, and let us illustrate this step on an example. Consider the following statement: $\mathsf{if}\ \langle s_1 \rangle_{\mathrm{b}}\ \mathsf{then}\ s_2\ \mathsf{else}\ s_3$. We replace it by $\mathsf{let\ mut}\ b_1 = \mathsf{false\ in}\ (s_1' \,;\, \mathsf{if}\ b_1\ \mathsf{then}\ s_2\ \mathsf{else}\ s_3)$, where s_1' is obtained by replacing boolean return statements ($\mathsf{return}\ b$) by assignments of the form ($\mathsf{if}\ b\ \mathsf{then}\ b_1 \leftarrow \mathsf{true\ else\ skip}$).

(E) *Elimination of Let Output Statements*, i.e., of statements of the form $\mathsf{let}\ x = e\ \mathsf{in}\ s$. This is done by textually replacing $\mathsf{let}\ x = e\ \mathsf{in}\ s$ by $s[x \mapsto e]$.
(F) *Elimination of Return Statements* for list expressions. First, to make sure that the program does not produce any output after the first return statement, we introduce a boolean variable `has_returned`, and guard every yield statement by a check on this variable. Then, we replace every statement $\mathsf{return}\ e$ by a for loop $\mathsf{for}^{\rightarrow}\ (i, x)\ \mathsf{in}\ e\ \mathsf{do}\ \mathsf{yield}\ x$. This is not possible if the return statement is of type List_0, and for this edge case, we refer the readers to our implementation.
(G) *Expansion of For Loops*, ensuring that every for loop iterates over a single list variable. This is the key step of the compilation, and it will be thoroughly explained later in this section.
(H) *Defining booleans at the beginning of for loops*. This is a technical step that ensures that all boolean variables are defined at the beginning of the program or at the beginning of a for loop. Thanks to the no-shadowing rule (R. V), we can safely move all boolean definitions to the top of their scopes.

Theorem 1. *The rewriting steps (Step A— Step H) all terminate and preserve typing. Moreover, normalized for-programs of type $\mathsf{List}_1 \to \mathsf{List}_1$ are isomorphic to simple for-programs.*

Forward For Loop Expansion. We now focus on the expansion of for loops, that is, Step G. The case of forward iterations is simpler and will illustrate a first difficulty. We replace each loop of the form $\mathsf{for}^{\rightarrow}\ (i, x)\ \mathsf{in}\ \langle s_1 \rangle_1\ \mathsf{do}\ s_2$ by the statement s_1 where every statement $\mathsf{yield}\ e$ is replaced by $s_2[x \mapsto e]$. This rewriting is problematic because it leaves the variable i undefined in s_2. The key observation allowing us to circumvent this issue is that the variable i can only be used in *comparisons*, and can only be compared with variables j that are iterating over $\langle s_1 \rangle_1$ (thanks to R. IV). It is therefore sufficient to order the outputs of s_1 to effectively remove the variable i from the program.

One can recover the ordering between outputs of s_1 by storing the position of the $\mathsf{yield}\ e$ responsible for the output, together with all position variables visible at that point. Let us illustrate this in a simple example:

$$(\mathsf{for}^{\leftarrow}\ (j, y)\ \mathsf{in}\ e\ \mathsf{do}\ (\underbrace{\mathsf{yield}\ y}_{p_1(j)}\ ;\ \underbrace{\mathsf{yield}\ \mathsf{char}(a)}_{p_2(j)}))\ ;\ \underbrace{\mathsf{yield}\ \mathsf{char}(b)}_{p_3}$$

In this example, there are three yield statements at positions p_1, p_2 and p_3. We can compute the *happens (strictly) before* relation between outputs of the various yield statements:

$$\mathsf{before}(p_1(j), p_2(j)) = \mathsf{true} \quad \mathsf{before}(p_2(j), p_3) = \mathsf{true} \quad \mathsf{before}(p_1(j), p_3) = \mathsf{true}$$

$$\mathsf{before}(p_1(j), p_1(j')) = j > j' \qquad \mathsf{before}(p_2(j), p_2(j')) = j > j'$$

$$\mathsf{before}(p_1(j), p_2(j')) = j \geq j'$$

In the case of $j = j'$, the output of $p_1(j)$ happens before the output of $p_2(j')$, because p_1 is the first yield statement in the loop. When $j > j'$, the output of $p_1(j)$ happens before the output of $p_2(j')$ because the loop is iterating in reverse order.

Backward For Loop Expansion. The case of backward iterations adds a new layer of complexity, namely to perform a non-reversible computation s in a reversed order: indeed, in the for loop $\mathsf{for}^{\leftarrow}\ (i, x)\ \mathsf{in}\ \langle s_1 \rangle_1\ \mathsf{do}\ s_2$, s_1 can contain the command $b \leftarrow \mathsf{true}$ which cannot be reversed.

Let us consider as an example $\mathsf{for}^{\leftarrow}\ (i, x)\ \mathsf{in}\ \langle s \rangle_1\ \mathsf{do\ yield}\ x$, where the statement s is defined to print all elements of a list u except the first one, namely:

$$s \ := \ \mathsf{let\ mut}\ b \ = \ \mathsf{false\ in\ for}^{\rightarrow}\ (j, y)\ \mathsf{in}\ u\ \mathsf{do\ if}\ b\ \mathsf{then\ yield}\ y\ \mathsf{else}\ \ b \leftarrow \mathsf{true}$$

The semantics of $\mathsf{for}^{\leftarrow}\ (i, x)\ \mathsf{in}\ \langle s \rangle_1\ \mathsf{do\ yield}\ x$ is to print all elements of u in reverse order, skipping the last loop iteration. To compute this new statement, we will use the following *trick* that can be traced back to [7, Lemma 8.1 and Fig. 6, p. 68]: we will use two versions of the statement s, the first one s_{rev}, will be s where all boolean introductions are removed, if statements $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ are replaced by sequences $s_1\ ;\ s_2$, every loop direction is swapped, and every sequence of statements is reversed. Its intended semantics is to reach all possible yield statements of s in the reversed order. In our case:

$$s_{\mathsf{rev}} \ := \ \mathsf{for}^{\leftarrow}\ (j', y')\ \mathsf{in}\ u\ \mathsf{do\ yield}\ y'$$

Some yield statements are reachable in s_{rev}, but not when iterating over s in reverse order. To ensure that we only output correct elements, we replace every $\mathsf{yield}\ \cdot$ statement in s_{rev} by a copy of s, leading to the programs $s' = s_{\mathsf{rev}}[\mathsf{yield}\ \cdot \mapsto s]$. In our case:

$$s' \ := \ \mathsf{for}^{\leftarrow}\ (j', y')\ \mathsf{in}\ u\ \mathsf{do}\ s$$

It is now possible to replace every yield statement in this new program by a conditional check ensuring that the output would actually be produced by the original program s.

$$s'' = s'[\mathsf{yield}\ e \mapsto \mathsf{if}\ i \ = \ j'\ \mathsf{then\ yield}\ e]$$

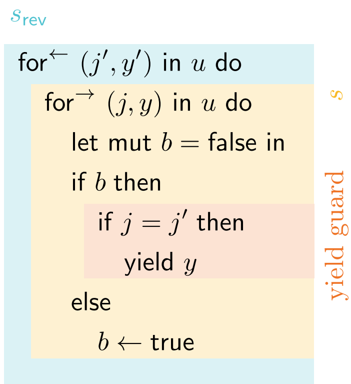

Fig. 5. Backward for loop expansion.

In our case, the final program is described in Fig. 5. This rewriting can be generalised to any program of the form $\mathsf{for}^{\leftarrow}\ (i, x)\ \mathsf{in}\ \langle s_1 \rangle_1\ \mathsf{do}\ s_2$ combining the construction illustrated here with the one taking care of position variables in the case of forward loops.

5 Simple For Programs and Interpretations

In this section, we show how to compile a simple for-program into a first-order interpretation in the symbolic setting. Recall that this is already known to be theoretically possible in the non-symbolic case [7]. However, this existing construction is not efficient: It requires computing a normal form of the simple for-program ([7, Lemma 5.2]), and goes through the model of pebble transducers [7, Section 5]—both of these steps significantly increase the complexity of the generated formulas.

To transform a simple for-program into a first-order interpretation, we use as transduction tags the set of all print statements in the program, remembering their location in the source code. The arity of a print statement is the number of the position variables present in its scope. The output function of a print statement is easy to define: if the print statement outputs a fixed character c, then the output function returns c; otherwise, if the print statement outputs $\mathsf{label}(i)$, then the output function returns the De Bruijn index [10] of the variable i. For the ordering formula between two print statements, we use the technique for comparing addresses of the print statements, described in the for loop expansion procedure: In order to compare of two print statements, we compare their shared position variables, breaking the ties using their ordering in the source code. Observe that such ordering formulas do not use quantifiers.

The hardest part is the domain formula. This difficulty is akin to the one of the for loop expansion procedure for the reverse loop: given a print statement $p(i_1, \ldots, i_k)$, where $i_1, \ldots, i_k$ are the position variables in the scope of the print, we need to check whether it can be reached. This amounts to taking the conjunction of the `if`-conditions, or their negations depending on the `if`-branch, along the path from the root of the program to the print statement. The only difficulty in defining this conjunction is using the first-order logic to compute the values of the boolean variables used in the `if`-conditions. We do this, by defining program formulas, which are first-order formulas that describe how a program statement transforms the values of its boolean variables.

5.1 Program Formulas

A *program formula* is a first-order formula where every free variable is either: an *input boolean variable* $\mathsf{in}_{\mathbb{B}}(b)$, an *output boolean variable* $\mathsf{out}_{\mathbb{B}}(b)$, or an *input position variable* $\mathsf{in}_{\mathbb{N}}(i)$. In order to accommodate the boolean variables, we introduce a new two-element sort $\mathbb{B}$. We handle it in the same way as the tag sort from Sect. 3.4.

Given a fixed word $w \in \mathcal{D}^*$, a program formula φ defines a relation between the input boolean variables $\mathsf{in}_{\mathbb{B}}(b_1), \ldots, \mathsf{in}_{\mathbb{B}}(b_n)$, input position variables $\mathsf{in}_{\mathbb{N}}(1), \ldots, \mathsf{in}_{\mathbb{N}}(k)$, and the output boolean variables $\mathsf{out}_{\mathbb{B}}(b_1), \ldots, \mathsf{out}_{\mathbb{B}}(m)$. We are only interested in the program formulas that define *functions* between the input and output variables, for every w.

In this section we show how to compute program formulas for every program statement s, that describes how the statement transforms its state. The formulas are constructed inductively on the structure of the statement. We start with the simplest case of `b := True`, whose program formula is defined as $\Phi_{\texttt{setTrue}} := \mathsf{out}_{\mathbb{B}}(b)$. Similarly, the program formula for a print statement is defined as $\Phi_{\texttt{print}} := \top$ (as it does not input or output any variables). For the induction step, we need to consider three constructions: conditional branching, sequencing, and iteration.

Conditional Branching. Given two program formulas Φ_1 and Φ_2 and a formula φ that only uses input variables (position and booleans), we simulate the `if then else` construction in the following way:

$$\Phi_{\texttt{if } \varphi \texttt{ then } \Phi_1 \texttt{ else } \Phi_2} := (\varphi \wedge \Phi_1) \vee (\neg\varphi \wedge \Phi_2) \quad .$$

This construction only works if Φ_1 and Φ_2 have the same output variables. If this is not the case, we can to extend Φ_1 and Φ_2 with identity on the missing output variables, by adjoining them with conjunctions of the form $\mathsf{in}_{\mathbb{B}}(b) \iff \mathsf{out}_{\mathbb{B}}(b)$ for each missing variable.

Composition of Program Formulas. Let us consider two program formulas Φ_1 and Φ_2, and denote their input and output boolean variables as $B_1^{\mathsf{in}}, B_1^{\mathsf{out}}$ and $B_2^{\mathsf{in}}, B_2^{\mathsf{out}}$. Let us start with the case where $B_2^{\mathsf{in}} = B_1^{\mathsf{out}} = \{b_1, \ldots, b_n\}$. In this case, we can compose the two program formulas in the following way:

$$\Phi_1; \Phi_2 := \exists_{b_1:\mathbb{B}} \cdots \exists_{b_n:\mathbb{B}} \quad \Phi_1[\mathsf{out}_{\mathbb{B}}(x) \mapsto x] \wedge \Phi_2[\mathsf{in}_{\mathbb{B}}(x) \mapsto x]$$

If the sets B_1^{out} and B_2^{in} are not equal, we can deal with it by first ignoring every output variable b of Φ_1 that is not consumed by Φ_2. Interestingly, this requires an existential quantification: $\Phi_1' := \exists_{b':\mathbb{B}} \Phi_1[\mathsf{out}_{\mathbb{B}}(b) \mapsto b']$. Then, for each variable b that is consumed by Φ_2 but not produced by Φ_1, we add the identity clause $(\mathsf{in}_{\mathbb{B}}(b) \iff \mathsf{out}_{\mathbb{B}}(b))$ to Φ_1' obtaining Φ_1''. After this modification, we can compose Φ_1'' and Φ_2 with no problems.

This definition of composition requires us to quantify over all variables form $B_1^{\mathsf{out}} \cup B_2^{\mathsf{in}}$, which influences the quantifier rank of the resulting program formula. In our implementation, we are a bit more careful, and only quantify over the variables from $B_1^{\mathsf{out}} \cap (B_2^{\mathsf{in}} \cup B_2^{\mathsf{out}})$, obtaining the following bound:

$$\mathsf{qr}(\Phi_1; \Phi_2) \leq \max(\mathsf{qr}(\Phi_1), \mathsf{qr}(\Phi_2)) + |B_1^{\mathsf{out}} \cap (B_2^{\mathsf{in}} \cup B_2^{\mathsf{out}})| \quad .$$

Iteration of Program Formulas. The most complex operation on program formulas is the iteration. We explain this on a representative case of a program formula Φ which has a single input position variable $\mathsf{in}_{\mathbb{N}}(i)$, and whose output boolean variables are the same as the input boolean variables ($B^{\mathsf{in}} = B^{\mathsf{out}}$).

Given a word $w \in \mathcal{D}^*$, evaluating a forward loop over i in the range 0 to $|w|$ amounts to the following composition:

$$\Phi[\mathsf{in}_{\mathbb{N}}(i) \mapsto 0]; \Phi[\mathsf{in}_{\mathbb{N}}(i) \mapsto 1]; \cdots; \Phi[\mathsf{in}_{\mathbb{N}}(i) \mapsto |w|] \quad , \tag{1}$$

The main difficulty is to compute this composition independently of the length of the word w, while keeping the formula and its quantifier rank small.

To that end, we observe that Φ uses a finite number of boolean variables, and that each of those variables can only be set to `True` once (R. VII). As a consequence, in the composition in Equation (1), there are at most $|B^{\mathsf{out}}|$ steps that actually modify the boolean variables. Based on this observation, one can *accelerate* the computation of the composition by guessing the sequence of those steps $(p_1, \ldots, p_{|B^{\mathsf{out}}|})$. The resulting program formula Φ^* is given below (we assume that Φ contains at least 3 boolean variables, and we denote their set as $\{b_1, \ldots, b_n\}$ – the cases for $n \leq 2$ are either analogous or trivial):

$$\Phi^* := \exists_{p_1 \leq \cdots \leq p_n : \mathbb{N}} \tag{2}$$

$$\exists_{\boldsymbol{b}_0, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{n+1} : \mathbb{B}^n} \tag{3}$$

$$\bigwedge_{1 \leq j \leq n} \Phi(p_j; \boldsymbol{b}_{j-1}; \boldsymbol{b}_j) \tag{4}$$

$$\bigwedge_{1 \leq j \leq n+1} \forall_{p_{j-1} \leq p \leq p_j : \mathbb{N}} \Phi(p; \boldsymbol{b}_{j-1}; \boldsymbol{b}_{j-1}) \tag{5}$$

$$\bigwedge_{1 \leq i \leq n} (\boldsymbol{b}_0)_i = \mathsf{in}_{\mathbb{B}}(b_i) \tag{6}$$

$$\bigwedge_{1 \leq i \leq n} (\boldsymbol{b}_{n+1})_i = \mathsf{out}_{\mathbb{B}}(b_i) \quad . \tag{7}$$

The structure of this formula is as follows: In Eq. (2), it guesses the steps $p_1, \ldots, p_n$ that actually modify the boolean variables. In Eq. (3), it guesses the intermediate values of the boolean variables ($\boldsymbol{b}_j$'s denote vectors of n boolean variables). In Eq. (4), it asserts that the guesses where *correct*, i.e., that the program formula Φ applied to position p_j and the boolean variables $\boldsymbol{b}_{j-1}$ produces the boolean variables $\boldsymbol{b}_j$. In Eq. (5), it ensures that no position different than the p_i's modifies the boolean variables. (In this equation, p_0 and p_{n+1} denote the first and the last position of the word.) Finally, in Eq. (6) and Equation (7), it ensures that the initial and final values of the boolean variables are correctly set to the input and output values. The formula for the reverse loop is similar, but guesses the positions p_i in a decreasing order.

Our construction ensures the following bound on the quantifier rank of the resulting program formula, which shows that the number of modified boolean variables is a crucial parameter for the complexity of the overall procedure:

$$\mathsf{qr}(\Phi^*) \leq \mathsf{qr}(\Phi) + |B^{\mathsf{out}}|^2 + |B^{\mathsf{out}}| + 1 \quad . \tag{8}$$

6 Implementation

We implemented all the transformations expressed in this paper in a `Haskell` program. To measure the complexity of these transformations, we associated to a high-level for-program the following parameters: its *size* (number of control flow statements), its *loop depth* (the maximum number of nested loops),

Table 1. Results for the transformations. Here FP is a for-program, S.FP is a simple for-program, and FO-I is a first-order interpretation. The columns **l.d.**, **b.d.** and **q.r.** stand respectively for the loop depth, boolean depth and quantifier rank.

filename	FP			S.FP			FO-I	
	size	l.d.	b.d.	size	l.d.	b.d.	size	q.r.
`identity.pr`	3	1	0	2	2	0	1	0
`reverse.pr`	3	1	0	2	2	0	1	0
`subwords_ab.pr`	24	2	1	15	4	3	956	14
`map_reverse.pr`	36	2	1	18	4	1	285	5
`prefixes.pr`	6	2	0	5	3	0	2	0
`get_last_word.pr`	18	1	1	23	4	2	8553	15
`get_first_word.pr`	22	1	1	5	2	0	103	4
`compress_as.pr`	12	1	1	12	3	2	209	10
`litteral_test.pr`	29	1	1	129	3	12	3.2×10^4	82
`bibtex.pr`	110	2	1	802	6	29	13.7×10^6	136

and its *boolean depth* (the maximum number of boolean variables visible at any point in the program). We compute the same parameters for the corresponding simple for-program. In the case of first-order interpretations, we only compute its *size* (number of nodes in the formula) and its quantifier rank. This allowed us to estimate the complexity of our transformations on a small set of programs that we present in Table 1. Then, we used several existing solvers to verify basic first-order Hoare triples for these programs. We illustrate in Table 2 the behaviour of the solvers on various verification tasks, with a timeout of 5 seconds for every solver. These test offer only initial insight into the performance of our implementation, so developing our implementation into an actual tool would require systematic benchmarks and comparison with already existing tools.

Table 2. Verification of first-order Hoare triples over sample for-programs. We specify the preconditions and postconditions as regular languages, writing $\mathcal{L}_{ab}$ as a shorthand for $\mathcal{D}^* ab \mathcal{D}^*$, and similarly for $\mathcal{L}_{aa}$, $\mathcal{L}_{ba}$, etc. In the columns corresponding to the solvers, a checkmark indicates a positive reply, a cross mark indicates a negative reply, and a question mark indicates a timeout or a memory exhaustion. We indicate the size and the quantifier rank (**q.r.**) of the first-order formulas that are fed to the solvers.

Name	Pre.	Post.	q.r.	size	MONA	CVC5	Z3
`compress_as.pr`	$\mathcal{L}_{ab}$	$\mathcal{L}_{ab}$	16	763	✓	?	?
`reverse_add_hash.pr`	$\mathcal{L}_{ab}$	$\mathcal{L}_{ba}$	9	380	?	✓	?
`get_last_word.pr`	$\mathcal{D}^* a$	$\mathcal{L}_{aa}$	27	28274	?	?	✗
`subwords_ab.pr`	$\mathcal{L}_{ab}$	$\mathcal{L}_{ab}$	26	3276	?	?	?
`map_reverse.pr`	$\mathcal{D}^* a$	$a\mathcal{D}^*$	13	801	?	?	?

Compilation to FO-Formulas. Looking at Table 1, we observe that the generated simple for programs have reasonable size and boolean depth. The generated first-order interpretations still have reasonable quantifier ranks, but their size grows significantly. In the simplest cases of Table 1, our compilation procedure is able to eliminate all boolean variables, thus producing a *quantifier-free* formula. This is the case for `identity.pr`, `reverse.pr` and `prefixes.pr`. Moreover, we observe that the boolean depth of the simple for-program is a good indicator of the quantifier rank of the generated first-order interpretation. Furthermore, the tests indicate that elimination of literals is responsible for a significant increase of the formulas size and quantifier rank (`literal_test.pr` and `bibtex.pr`). This is explained by the fact that the elimination of literals introduces (non-cyclic) counters, simulated by a number of boolean variables. Finally, we observe that the size of the generated formulas differs significantly for the programs `get_first_word.pr` and `get_last_word.pr`. This is somewhat surprising, as the two programs are symmetric with respect to reversing the input words, and indicates some room for improvement in handling the reversed iteration.

Solver Performance. We can observe in Table 2 that the different solvers are complementary. This might seem surprising, as the MONA solver is a complete decision procedure. However, since it solves a problem that is TOWER-complete [27, Theorem 13.5], it is understandable that it underperforms the SMT solvers on some instances, even though we use them with the undecidable `UFDTLIA` theory. Let us justify this choice of the SMTLib theory: (a) *Uninterpreted Functions* (`UF`) are used to represent the word, which is treated as a function from positions to characters, (b) *Data Types* (`DT`) is used to represent finite sets of tags and characters, (c) *Linear Integer Arithmetic* (`LIA`) is used to deal with the order of the positions in the word. This choice might not be optimal, but we believe that it is a good trade off between ease-of-use and performance for our proof-of-concept implementation. We can also observe that no solvers was able to deal with `subwords_ab.pr` and `map_reverse.pr` within the 5 seconds timeout. Understanding the complexities that arise in those cases, might be helpful for improving the performance of our implementation.

7 Conclusion

We have show that the theory of star-free polyregular functions can be used to verify close to real-world programs, and have implemented a prototype tool that can discharge simple verification goals to existing solvers.

Benchmarks. It would be interesting to systematically benchmark our implementation against existing tools for verifying linear regular functions, in the case of first order specifications. Since our approach allows for polynomial size transformations, it would also be interesting to devise a set of benchmarks for this broader classe of functions.

Optimizations. The preliminary tests indicate that one of the most promising source of optimizations is managing the boolean depth of the generated simple for-programs during compilation. This can be achieved by post-compilation optimizations (constant propagation, dead code elimination), or by improving the code generation mechanism itself, which are low-hanging fruits for future work. One source of the boolean variables seems to be the elimination of Literal Equality step (Step B), which could be mitigated by adding explicit successor and predecessor predicates to the language of simple for-programs.

At the level of first-order interpretations, we have identified several directions for improving their efficiency. One optimization is computing the sequential composition of programs in a way that minimizes the number of quantified boolean variables. Similarly, there seems to be potential for performing direct substitutions instead of quantifying over the variables in a lot of cases. Finally, our current approach for handling loops introduces universal quantifiers, whose number could be reduced by exploiting the monotonicity of the state transformations.

Solver Integration. There is a lot of potential for optimizing the input and parameters of the solvers for our particular use-case. An interesting research direction would be to reduce the verification problem to emptiness of LTL formulas, allowing us to use LTL solvers such as `SPOT` [14].

Modular Verification. The benchmarks show that one of the main bottlenecks of our approach is the expansion of loops (whether in the translation to simple for-programs or in the translation to first-order interpretations). For this reason, the ability to verify statements of the form `for (i, e) in enumerate(f(x)) do s done`, based on a specification of f given as a Hoare triple, would be a significant improvement. However, it remains unclear how to integrate such modular verification in our current approach.

Language Design. As mentioned in Sect. 2, for-programs extended with unrestricted booleans also enjoy a decidable verification of Hoare triples. However, the verification algorithm uses of monadic second-order logic (MSO) over words instead of first-order logic. While this prohibits the use of traditional SMT solvers, this logic can be handled by the MONA solver, and it might be interesting to implement and test the unrestricted version of the language.

Another interesting extension of the language would be to allow the use of complex types, such as pairs and records. This would make the language closer to real use cases such as configuration management and data processing. It would require extending the specification language to structured data types, bypassing the current limitation that we can only verify string-to-string transformations.

Integration with Existing Tools. It would be a natural next step to integrate our results inside frameworks for program verification or testing. This could be by checking goals generated by a tool such as `Why3` [4], or by verifying properties of Python programs using decorated functions. We would also like to point out that verification methods based on a regularity preserving property (such as done in [13]) can transparently use our more general class of programs as input, instead of the more traditional linear regular functions.

Acknowledgements. We would like to thank Arnav Garg and Ojas Maheshwari for their participation in the early stage of this project.

References

1. Alur, R.: Streaming string transducers. In: Beklemishev, L.D., de Queiroz, R. (eds.) WoLLIC 2011. LNCS (LNAI), vol. 6642, pp. 1–1. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20920-8_1
2. Alur, R., Černý, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pp. 599–610. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/1926385.1926454
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland, pp. 53–64 (2011). https://hal.inria.fr/hal-00790310
5. Bojańczyk, M.: On the growth rates of polyregular functions. In: 2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–13. IEEE (2023)
6. Bojańczyk, M., Kiefer, S., Lhote, N.: String-to-string interpretations with polynomial-size output. arXiv preprint arXiv:1905.13190 (2019)
7. Bojańczyk, M.: Polyregular functions (2018). https://arxiv.org/abs/1810.08760v1
8. Bojańczyk, M.: Slightly infinite sets (2019). https://www.mimuw.edu.pl/~bojan/paper/atom-book
9. Bojańczyk, M., Daviaud, L., Krishna, S.N.: Regular and first-order list functions. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '18, pp. 125–134. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3209108.3209163
10. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5), 381–392 (1972). https://doi.org/10.1016/1385-7258(72)90034-0
11. Büchi, J.R.: Weak second-order arithmetic and finite automata. Math. Log. Q. **6**(1–6), 66–92 (1960). https://doi.org/10.1002/malq.19600060105
12. Carton, O., Dartois, L.: Aperiodic two-way transducers and FO-transductions. In: Kreutzer, S. (ed.) 24th EACSL Annual Conference on Computer Science Logic (CSL 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 41, pp. 160–174. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2015). https://doi.org/10.4230/LIPIcs.CSL.2015.160

13. Chen, T., Hague, M., Lin, A.W., Rümer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290362. http://www.philipp.ruemmer.org/publications/atva2020.pdf
14. Duret-Lutz, A., et al.: From Spot 2.0 to Spot 2.10: what's new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22). Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-13188-2_9
15. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
16. Engelfriet, J., Maneth, S.: Two-way finite state transducers with nested pebbles. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 234–244. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45687-2_19
17. Filiot, E., Reynier, P.A., Lhote, N.: Lexicographic transductions of finite words (2025). https://arxiv.org/abs/2503.01746
18. Jeż, A., Lin, A.W., Markgraf, O., Rümmer, P.: Decision procedures for sequence theories. In: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II, CAV'23, pp. 18–40. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_2. https://anthonywlin.github.io/papers/cav23.pdf
19. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
20. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, University of Aarhus (2001), notes Series NS-01-1. http://www.brics.dk/mona/
21. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata Stud. **34**, 3–42 (1956). https://doi.org/10.1515/9781400882618-002
22. McNaughton, R., Papert, S.A.: Counter-Free Automata. The MIT Press, Cambridge (1971). https://doi.org/10.5555/1097043
23. Microsoft Research: Z3 theorem prover (2008). https://github.com/Z3Prover/z3, swh:1:dir:d6e5b24a3751b89fb1a4844c5a80a6393c2c1fa6
24. Muscholl, A., Puppis, G.: The many facets of string transducers. In: Niedermeier, R., Paul, C. (eds.) 36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 126, pp. 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2019). https://doi.org/10.4230/LIPIcs.STACS.2019.2. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2019.2
25. Perrin, D., Pin, J.É.: First-order logic and star-free sets. J. Comput. Syst. Sci. **32**(3), 393–406 (1986). https://doi.org/10.1016/0022-0000(86)90037-1. https://www.sciencedirect.com/science/article/pii/0022000086900371
26. Rabin, M.O., Scott, D.S.: Finite automata and their decision problems. IBM J. Res. Dev. **3**(2), 114–125 (1959). https://doi.org/10.1147/RD.32.0114
27. Reinhardt, K.: The complexity of translating logic to finite automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata Logics, and Infinite Games. LNCS, vol. 2500, pp. 231–238. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4_13
28. Schützenberger, M.P.: On the definition of a family of automata. Inf. Control **4**(2–3), 245–270 (1961). https://doi.org/10.1016/S0019-9958(61)80020-X
29. Schützenberger, M.P.: On finite monoids having only trivial subgroups. Inf. Control **8**(2), 190–194 (1965). https://doi.org/10.1016/S0019-9958(65)90108-7

30. Trakhtenbrot, B.A.: Finite automata and the logic of one-place predicates. Am. Math. Soc. Translat. **59**(2), 23–55 (1966). https://doi.org/10.1090/trans2/059/02
31. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993). https://doi.org/10.7551/mitpress/3054.001.0001
32. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G., et al. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.