Synthesizing accumulative functions via program transformation

Junyu Lin[1] and Akimasa Morihata[2]

[1] The University of Tokyo, Komaba 3-8-1, Meguro, Tokyo, Japan
lin-junyu108@g.ecc.u-tokyo.ac.jp
[2] The University of Tokyo, Komaba 3-8-1, Meguro, Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp

Abstract. Accumulative functions, such as tail-recursive functions, employ accumulation parameters to carry and update the intermediate results. Despite their ubiquity and importance for efficient implementations, the automatic synthesis of accumulative functions remains challenging. The presence of accumulative parameters not only expands the search space but also unfastens the input-output examples from the traces of recursive calls, leading existing program synthesis methods to either fail in generating nontrivial accumulative functions or rely on pre-provided skeletons of recursive calls with accumulations. In this paper, we investigate an alternative approach to synthesizing accumulative functions. Our strategy integrates an off-the-shelf synthesizer, which may not inherently produce accumulative functions, and a program transformation that derives accumulative functions from non-accumulative ones. We specifically focus on the transformation introduced by Kühnemann et al. (RTA 2001), which effectively derives accumulative functions if the non-accumulative ones consist of substitution operators. By guiding the synthesizer to use substitution operators, we aim to obtain functions suitable for the transformation. We demonstrate the ability of our approach with examples from existing benchmarks.

Keywords: Program synthesis · Accumulative function · Program transformation.

1 Introduction

Accumulative functions refer to a type of function that uses an accumulator to keep track of and update the intermediate results as part of a computation. A typical example is the tail-recursive function. The following *tail_reverse* serves as a tail-recursive, linear-time variant of the non-accumulative quadratic-time *reverse* function, using the list concatenation function *append*.

$$
\begin{aligned}
\mathit{reverse}\ [] &= [] \\
\mathit{reverse}\ (x :: \mathit{xs}) &= \mathit{append}\ (\mathit{reverse}\ \mathit{xs})\ (x :: []) \\
\mathit{append}\ []\ y &= y \\
\mathit{append}\ (x :: \mathit{xs})\ y &= x :: \mathit{append}\ \mathit{xs}\ y
\end{aligned}
$$

$$
\begin{aligned}
\mathit{tail\_reverse}\ x &= \mathit{tail\_reverse}_1\ x\ [] \\
\mathit{tail\_reverse}_1\ []\ y &= y \\
\mathit{tail\_reverse}_1\ (x :: \mathit{xs})\ y &= \mathit{tail\_reverse}_1\ \mathit{xs}\ (x :: y)
\end{aligned}
$$

In recent years, recursive function synthesis has become a significant research area [1, 6, 8, 11, 13–16, 21]. Some methods leverage the programming-by-example (PBE) paradigm due to the simplicity and accessibility of input-output examples for specifying program behavior. These methods generally depend on recursive traces, and some of them [1, 16] even require *trace-complete* examples, consisting of all recursive calls' input-output pairs. For example, consider a set of input-output examples for the *reverse* function, $\{[1,2] \to [2,1], [2] \to [2], [] \to []\}$, where $x \to y$ denotes inputting x to the function must result in y. This is trace-complete, as it contains all recursive calls' information of *reverse* $[1,2]$. Trace-complete examples allow candidate expressions to be searched directly. For instance, from the input-output examples above, we can immediately know *reverse* $[] = []$; moreover, we can easily guess that the right-hand side for *reverse* $[1,2]$ is an expression to calculate $[2,1]$ from 1 and $[2]$, since the sub-computation, *reverse* $[2]$ results in $[2]$. Others [13, 21] attempt to reconstruct complete traces during synthesis.

However, accumulative function synthesis remains challenging. For example, the *tail_reverse* function involves the auxiliary recursive function, $\mathit{tail\_reverse}_1$, whose synthesis may necessitate its trace-complete examples with the accumulator y, say $\{([1,2],[]) \to [2,1], ([2],[1]) \to [2,1], ([],[2,1]) \to [2,1]\}$, as well as the initial accumulator value. However, such input-output examples for the auxiliary function are not natural to prepare. It is desirable to synthesize accumulative functions from the input-output examples for the top-level function, namely *tail_reverse*.

Some synthesizers [6, 8, 14] can produce accumulative functions with a user-provide recursion pattern, which is either a recursion skeleton [6], or a partial implementation with holes, called program sketching [8, 14]. For example, we can synthesize *tail_reverse* with the following sketch in which the shadow parts are holes.

$$
\begin{aligned}
\mathit{tail\_reverse}\ x &= \mathit{tail\_reverse}_1\ x \\
\mathit{tail\_reverse}_1\ []\ y &= \\
\mathit{tail\_reverse}_1\ (x :: \mathit{xs})\ y &= \mathit{tail\_reverse}_1\ \mathit{xs}
\end{aligned}
$$

However, preparing an appropriate sketch is in general non-trivial. A sketch with a too-large hole will fail in the synthesis, whereas one specifying a too-detailed

computation structure is less satisfactory. We can hardly know in advance what level of detail is necessary for a successful synthesis.

In this paper, we address the synthesis of accumulative functions, considering an alternate approach via program transformation. We focus on the method by Kühnemann et al. [12], which can derive an accumulative function by eliminating a special kind of operator, named substitution operator. We begin the synthesis process with preparing a set of substitution operators. Next, we instruct the synthesizer to construct a non-accumulative recursive function consisting of the substitution operator. Finally, we transform the obtained function into accumulative versions. In our approach, the synthesizer avoids the complexity of introducing an accumulator and does not rely on a pre-provided program skeleton.

To evaluate the promise of our approach, we collected benchmark examples from a state-of-the-art synthesis approach PARA [8] and conducted an experiment. We employed TRIO [13], which is a state-of-the-art PBE-based synthesizer, as the underlying synthesizer. Our approach could synthesize 8 out of 10 accumulative versions of the benchmark functions that can be optimized by accumulation with 10 pairs of input-output examples. The existing sketching-based synthesizer, SMYTH [14], was less successful. It succeeded in only 2. Even with a sketch containing an appropriate base-case expression and initial accumulation value, it could synthesize only 6. The experiments show that our approach is promising in accumulative function synthesis.

In summary, our paper makes the following contributions.

- We investigate an approach that synthesizes an accumulative function via program transformation. The approach consists of a synthesis of a non-accumulative program and a transformation into an accumulative one.
- We take examples from existing benchmark programs to demonstrate the ability of our approach. The result shows our approach can synthesize natural accumulative functions for several examples.

2 Background

2.1 Modular Tree Transducer

Kühnemann et al. introduced a transformation method based on modular tree transducers (ModT) [5]. Modular tree transducer is an extension of macro tree transducer (MTT) [4]. A macro tree transducer can be considered as a recursive first-order functional program over trees. We can define an MTT as below. We assume that f is a function name, c is a tree constructor, x_i, y_i are variables and r is the right-hand side. We only consider total deterministic MTTs.

$$
\begin{aligned}
\mathit{prog} &\coloneqq \mathit{decl} \cdots \mathit{decl} \\
\mathit{decl} &\coloneqq f\ (c\ x_1 \cdots x_n)\ y_1 \cdots y_m = r \\
r &\coloneqq x_i \mid y_i \mid c\ r \cdots r \mid f\ x_i\ r \cdots r
\end{aligned}
$$

An n-modular tree transducer (n-ModT) is a hierarchy $(m_1, \ldots, m_n)$ of modules. Each module m_i $(i \leq n)$ forms an MTT except that its function may call functions defined in the modules $m_{i+1}, \ldots, m_n$ as if they are constructors.

From the definition, 1-ModT is an MTT, and *reverse* is a 2-ModT since it contains 2 modules: *reverse* and *append*. And *tail_reserve* is an MTT because it consists of only the *tail_reserve* module.

2.2 Substitution function and *Yield* function

A substitution module consists of substitution functions and provides a mechanism for substituting accumulation parameters with some leaves. Let $\pi_1, \ldots, \pi_n$ be a series of distinct constructors. Given $i, j \in \mathbb{N}$ and $j \leq i \leq n$, sub_i is a substitution function defined below, where c is a constructor other than $\pi_1, \ldots, \pi_i$.

$$sub_i\ \pi_j\ y_1\ \cdots\ y_i = y_j \tag{1}$$

$$sub_i(c\ x_1\ \cdots\ x_k)\ y_1\ \cdots\ y_i = c\ (sub_i\ x_1\ y_1\ \cdots\ y_i)\ \cdots\ (sub_i\ x_k\ y_1\ \cdots\ y_i) \tag{2}$$

For example, recall the *append* module defined below.

$$\begin{aligned} append\ []\ y &= y \\ append\ (x :: xs)\ y &= x :: append\ xs\ y \end{aligned}$$

It forms a substitution module, where $append = sub_1$ and $\pi_1 = []$[3].

A *Yield* module translates an expression in which the tree substitution operations are symbolically represented, into the tree it denotes. The following is the definition of each function, $Yield_n$ $(n \in \mathbb{N})$, in a *Yield* module. We use sub_i to denote the symbolic representation of the substitution as well so as to make its connection to substitution functions obvious.

$$Yield_n\ \pi_i\ y_i \cdots y_n = y_i \tag{3}$$

$$Yield_n\ (c\ x_1 \cdots x_k)\ y_1\ \cdots\ y_n = c\ (Yield_n\ x_1\ y_1 \cdots y_n) \cdots (Yield_n\ x_k\ y_1 \cdots y_n) \tag{4}$$

$$Yield_n\ (sub_m\ x\ y_1 \cdots y_m)\ {y'}_1 \cdots {y'}_n = Yield_m\ x\ (Yield_n\ y_1\ y'_1 \cdots y'_n) \cdots (Yield_n\ y_m\ y'_1 \cdots y'_n) \tag{5}$$

2.3 Transformation Algorithm

Based on the definitions above, the transformation by Kühnemann et al. composes the two modules of a 2-ModT (m_1, m_2) to an MTT if m_2 is a substitution

[3] Strictly speaking, we must regard $(x ::\ )$ as a constructor; otherwise *append* must traverse x as well, which looks like containing a type error. We disregard this kind of minor adjustment between ModTs and functional programs.

module and m_1 is non-accumulative. We will take *reverse* defined below as an example to explain their approach.

$$\begin{aligned} \mathit{reverse}\ [] &= [] \\ \mathit{reverse}\ (x :: \mathit{xs}) &= \mathit{append}\ (\mathit{reverse}\ \mathit{xs})\ (x :: []) \end{aligned}$$

Freezing The first step is preparing a new function f' (here, *reverse*$'$) by substituting the substitution functions (here, *append*) with new constructors sub_1. We call this step freezing and the new constructors frozen functions.

$$\begin{aligned} \mathit{reverse}'\ [] &= [] \\ \mathit{reverse}'\ (x :: \mathit{xs}) &= \mathit{sub}_1\ (\mathit{reverse}'\ \mathit{xs})\ (x :: []) \end{aligned}$$

Introducing *Yield* module After freezing, *reverse*$'$ becomes an MTT but sub_1 does not perform the "append" operation. To recover the original computation, we introduce the *Yield* module defined as follows.

$$\begin{aligned} \mathit{Yield}_0\ [] &= [] \\ \mathit{Yield}_0\ (\mathit{sub}_1\ \mathit{xs}\ y) &= \mathit{Yield}_1\ \mathit{xs}\ y \\ \mathit{Yield}_1\ []\ y &= y \\ \mathit{Yield}_1\ (\mathit{sub}_1\ \mathit{xs}\ y)\ y' &= \mathit{Yield}_1\ \mathit{xs}\ (\mathit{Yield}_1\ y\ y') \end{aligned}$$

Note $\mathit{reverse}\ \mathit{xs} = \mathit{Yield}_0\ (\mathit{reverse}'\ \mathit{xs})$ holds from the construction of *reverse*$'$.

Deforestation $\mathit{Yield}_0\ (\mathit{reverse}'\ \mathit{xs})$ is still inefficient because the *Yield* module recursively traverses and reconstructs the tree generated by *reverse*$'$. We eliminate this inefficiency by applying deforestation [20] that eliminates intermediate lists and trees.

For example, we compose the second line of *reverse*$'$ and Yield_i functions via the unfolding-folding approach [3]. Assume composing *reverse*$'$ and Yield_i is $\mathit{reverse}_i$.

$$\begin{aligned} \mathit{reverse}_1\ (x :: \mathit{xs})\ y &= \mathit{Yield}_1\ (\mathit{reverse}'\ (x :: \mathit{xs})\ y) \\ &= \mathit{Yield}_1\ (\mathit{sub}_1\ (\mathit{reverse}'\ \mathit{xs})\ (x :: []))\ y \\ &= \mathit{Yield}_1\ (\mathit{reverse}'\ \mathit{xs})\ (\mathit{Yield}_1\ (x :: [])\ y) \\ &= \mathit{reverse}_1\ \mathit{xs}\ (x :: y) \end{aligned}$$

Other parts are obtained similarly. Eventually, we get the following accumulative function.

$$\begin{aligned} \mathit{reverse}_0\ [] &= [] \\ \mathit{reverse}_0\ (x :: \mathit{xs}) &= \mathit{reverse}_1\ \mathit{xs}\ (x :: []) \end{aligned}$$

$$\begin{aligned} \mathit{reverse}_1\ []\ y &= y \\ \mathit{reverse}_1(x :: \mathit{xs})\ y &= \mathit{reverse}_1\ \mathit{xs}\ (x :: y) \end{aligned}$$

Compared with the original function, this function has a linear time complexity. Hence, we successfully derive an efficient accumulative variant of *reverse*.

3 Accumulative function synthesis via transformation

3.1 Experiment setup

We implemented our approach as follows. We first inputted the definitions of substitution operators and input-output examples (10 pairs) to TRIO [13] to obtain a non-accumulative recursive function. Then, we manually applied the transformation described in Section 2 if possible.

For comparison, we select SMYTH [14], which is one of the most famous template-based synthesizers. It can generate accumulative programs if an accumulative partial program (sketch) is given. We inputted the same substitution operators and input-output examples as the case of TRIO. At this step, we only inputted the top definition of the function and initialization of the accumulator as the program sketch for SMYTH. If SMYTH failed to generate functions with this sketch, we modified the sketch by adding base-case expressions.

In either case, the synthesis is regarded as a failure when the synthesizer executes time out (max 300 seconds) or outputs wrong answers. In our approach, outputting a non-accumulative function for which the transformation is inapplicable is also a failure.

3.2 Benchmark examples

We exhaustively picked up examples for which accumulative implementations are natural and efficient from the benchmark used in the state-of-the-art study [8] and obtained 10 functions. They can be divided into three categories: three natural number functions (`nat_mul`, `nat_exp`, and `nat_factorial`), two list functions (`list_sum` and `list_reverse`), and five tree functions (`tree_preorder`, `tree_inorder`, `tree_postorder`, `tree_count_leaves`, and `tree_count_nodes`).

3.3 Experimental results

Table 1 summarizes the results of our experiments. "operator" means the substitution operators we inputted. ✓ denotes the method successfully synthesizes an accumulative version of the target function. ✗ denotes the failure of synthesis which is time-out or outputting the wrong answer. ✔ means the method fails to synthesize with the original setup but succeeds after the modification of the condition.

Our approach could synthesize 8 out of 10 benchmark functions but was time-out in `nat_exp` and `nat_factorial` with the original setup. SMYTH successfully generated 6 benchmark functions when the base-case pattern was determined and only succeeded 2 functions without the undetermined base-case pattern.

In what follows, we pick one example for each category and analyze it in detail. For better readability, we avoid showing cryptic programs obtained from the synthesizers and omit the input-output examples.

Table 1. Result of experiments

function (operator)	**our approach**	**SMYTH** (base-case)	**SMYTH** (no base-case)
`nat_mul` (`add`)	✓	✓	✗
`nat_exp` (`add`, `mul`)	✗	✗	✗
`nat_factorial` (`add`, `mul`)	✔	✗	✗
`list_sum` (`add`)	✓	✓	✓
`list_reverse` (`append`)	✓	✓	✓
`tree_preorder` (`append`)	✓	✓	✗
`tree_inorder` (`append`)	✓	✓	✗
`tree_postorder` (`append`)	✓	✓	✗
`tree_count_leaves` (`add`)	✓	✗	✗
`tree_count_nodes` (`add`)	✓	✗	✗

Natural number functions Here we pick `nat_mul`, whose objective is to multiply two natural numbers The substitution operator `add` below is supplied to the synthesizers.

```
let rec add m n =
    match m with
    | Z -> n
    | S m1 -> S (add m1 n)
```

TRIO could generate the following program.

```
let rec f x y =
    match x with
    | Z -> y
    | S x1 -> add (f x1 y) y
```

Then eliminating `add` makes it accumulative. In this function, `a` is the accumulator, increasing `y` in every recursive call.

```
let rec f0 x y =
  match x with
    | Z -> x
    | S n -> f1 n y y

and f1 x y a =
  match x with
    | Z -> a
    | S n -> f1 n y (add y a)
```

SMYTH could synthesize the accumulative `nat_mul` from the following sketch, in which `??` denotes the hole. Note that SMYTH failed if the whole body of `f1` is abstracted to a hole.

```
let f0 x y =
  let rec f1 x y a =
```

```
    match x with
      | Z ->
          a
      | S n -> ??
          (* f1 n y (add y a) *)
  in f1 x Z
```

List functions To synthesize the `list_reverse` function, we specified `append` as a substitution operator.

```
let rec append l1 l2 =
    match l1 with
    | [] -> l2
    | Cons (h,t) -> Cons(h, append t l2)
```

Then TRIO outputted the non-accumulative `list_reverse`.

```
let rec f x =
    match x with
    | [] -> x
    | Cons (x, xs) -> append (f xs) (Cons(x, []))
```

After applying the transformation, we obtain the accumulative version of `list_reverse`.

```
let rec f0 x =
  match x with
    | [] -> []
    | Cons (x, xs) -> f1 xs (Cons(x, []))
and
f1 x y =
  match x with
    | [] -> y
    | Cons (x, xs) -> f1 xs (Cons(x, y))
```

An accumulative `list_reverse` can be synthesized by SMYTH from the following sketch, which contains the initial accumulation value but does not the base-case expression.

```
let f0 x =
  let rec f1 x y = ??
    (* match x with
     *  | [] -> y
     *  | Cons (x, xs) -> f1 xs (Cons(z, y)) *)
  in f1 x []
```

Tree functions We take `tree_preorder` function as an example on tree functions. We inputted the specification with a substitution operator `append` as the same as the case of `reverse`.

TRIO outputted the following program.

```
let rec f x =
    match x with
    | Leaf -> []
    | Node(l, v, r) -> append (Cons(v, (f l))) (f r)
```

Then, we eliminate `append` to obtain accumulative `tree_preorder`.

```
let rec f0 x =
  match x with
  | Leaf -> []
  | Node(l, v, r) -> Cons (v, (f1 l (f0 r)))

and f1 x y =
  match x with
  | Leaf -> y
  | Node(l, v, r) -> Cons (v, (f1 l (f1 r y)))
```

SMYTH can fill the hole when the sketch contains the base-case expression in addition to the initial accumulation value.

```
let f0 x =
  let rec f1 x y =
    match x with
      | Leaf -> y
      | Node(l, v, r)-> ??
            (* Cons (v, (f1 l (f1 r y))) *)
  in f1 x []
```

3.4 Discussions

SMYTH failed in most examples if the sketch did not contain the base-case expression. This fact indicates that preparing an appropriate sketch is generally nontrivial. For example, the base-case expressions for `count_tree_nodes` and `count_tree_leaves` are different. In contrast, substitution operators are shared in most of our benchmark examples, being only three in total.

We can obtain accumulative functions by supplying the synthesizer with higher-order functions that specify the recursion pattern. In fact, supplying `fold` (a.k.a., `fold_left` or `foldl`) enables TRIO or SMYTH to synthesize `list_sum` and `list_reverse`. However, there are many such higher-order functions, including those for left-to-right, right-to-left, top-down, etc., and specifying one of them is, roughly speaking, at least more informative than specifying the sketch. Therefore, this higher-order-function-based approach inherits the difficulty of preparing an appropriate sketch.

The `nat_exp` and `nat_factorial` functions rely on the multiplication operator, `mul`. Unfortunately, `mul` is not the substitution operator on natural numbers because `mul` x (`S` y) $\neq$ `S` (`mul` x y). Hence, naively applying the transformation eliminates only `add`, which is the subcomputation of `mul`, and makes only

`mul`, i.e., neither `nat_exp` nor `nat_factorial`, accumulative. To synthesize accumulative `nat_exp` and `nat_factorial`, we must regard `mul` as a substitution operator. In fact we can, as `mul` satisfies the following equations.

$$\texttt{mul}\ (\texttt{S}\ \texttt{Z})\ z = z$$
$$\texttt{mul}\ (\texttt{add}\ x\ y)\ z = \texttt{add}\ (\texttt{mul}\ x\ z)\ (\texttt{mul}\ y\ z)$$

That is, `mul` is a substitution operator if `add` and `S Z` are the constructors. This observation reminds us of the study by Voigtländer [18]. He proposed a method of eliminating `reverse` and `map` in addition to `append`. Further study, including characterization of eliminatable operators and integration to the program synthesis, is a future work.

4 Related work

4.1 Recursive program synthesis

Recently, many approaches for recursive program synthesis have been proposed. TRIO [13] proposes two techniques for synthesizing recursive programs. The block-based pruning enables us to reject imperfect candidates of the recursive program by checking consistency with blocks (non-recursive expressions). The library sampling accelerates synthesis using library functions (i.e. user-provided external operators) by sampling their behaviors. SYRUP [21] uses recursion traces to build a trace-indexed version space algebra and select candidates by trace-based ranking function. BURST [15] firstly uses angelic semantics to enable bottom-up synthesis of recursive functions. Despite these impressive developments, they cannot synthesize accumulative recursive functions unless an accumulative recursion pattern is explicitly specified.

Among others, we selected TRIO as the underlying synthesizer of our approach mainly because of its excellent capability of user-defined external operators, which our approach must instruct the synthesizer to use. TRIO naturally accepts even complicated external operators, including those defined as recursive functions, as a part of the specification of the synthesis. Moreover, its library-sampling approach may reduce the cost of dealing with complex external operators. In addition, we anticipate that search-based synthesizers (including TRIO) might be preferable to constraint-solving-based ones. In the former, defining an appropriate search precedence may enable us to prioritize the use of substitution operators. In contrast, in the latter, the synthesis process is usually hidden behind the constraint solver, making it more difficult to control. We leave detailed analysis and experiments on this issue as future work.

Nevertheless, our approach is not specialized in TRIO and can use BURST and SYRUP, for example. However, adopting SYRUP is not very straightforward because SYRUP requires defining the external operator as a logical formula understandable by the underlying SMT solver.

CYPRESS [11] synthesizes recursive programs by cyclic program synthesis approach, which can discover recursive auxiliaries based on cyclic proofs.

Auxiliaries, such as `tail_reverse1` for `tail_reverse`, are often essential for developing accumulative programs. However, because the main specification of CYPRESS limits auxiliary recursive procedures without extra parameters, CYPRESS is unable to synthesize programs with accumulator parameters.

SYNDUCE [6] synthesizes a recursive function from a reference implementation that works on different data types. SYNDUCE uses a recursion template to limit the recursion strategy. It can generate accumulative functions by supplying an appropriate template. SMYTH [14] is also a template-based program synthesizer. Given a partial program with holes, i.e. a *template*, it finds the program fragment to fill the holes with iterative forward-backward abstract interpretation obtained by computing possible output (input) with corresponding input (output). It can generate accumulative programs if an accumulative template is given.

PARA [8] is another sketch-based synthesizer that uses paramorphisms. Paramorphism is a recursive pattern that allows you to not only fold a data structure but also access the rest of the original data structure during the recursion. Similar to SMYTH, it finds program fragments to fill holes in the preprovided template. Paramorphisms and accumulation appear to be orthogonal: the former enables extracting information from remaining unprocessed elements, whereas the latter focuses on maintaining additional information about elements already processed. Detailed comparison and cooperation is future work.

4.2 Accumulation

Transformational developments of accumulative functions, called *accumulation*, have been studied for a long time. They may be categorized into two approaches.

One approach [9, 17] aims at manually (or semi-automatically) driving nontrivial programs and algorithms, including the pioneering work by Bird [2]. This approach is generally difficult to coordinate with program synthesizers because it provides less feedback on what kind of programs the accumulation is more likely to succeed.

The other approach, sometimes called *deforestation* [20], eliminates intermediate data structures passed between functions. It is known that deforestation concerning some kinds of functions, the list concatenation operator in particular, leads to accumulative programs [7, 10, 19]. The transformation by Kühnemann et al. [12], which we have employed, generalizes the observation to arbitrary substitution operators. It is better suited for cooperation with program synthesizers because prioritizing substitution operators will likely lead to successful accumulation. It is worth noting that we can use a further generalization developed by Voigtländer [18], which can eliminate the reverse and map functions in addition to the substitution operators.

5 Conclusion and future works

In this paper, we present an alternative approach to synthesizing accumulative recursive programs using the transformation developed by Kühnemann et al.

Our method avoids the complexities inherent in directly synthesizing accumulative programs. The experiment shows that our approach can be an alternative approach to obtain accumulative functions.

Currently, we have little considered how to guide the synthesizer to generate the function that is suitable for transformation (for example, prioritize the use of substitution operators). We plan to investigate a system for supporting integration of program synthesis and transformations.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive Program Synthesis. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 934–950. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
2. Bird, R.S.: The Promotion and Accumulation Strategies in Transformational Programming. ACM Trans. Program. Lang. Syst. **6**(4), 487–504 (1984). https://doi.org/10.1145/1780.1781
3. Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the ACM **24**(1), 44–67 (jan 1977). https://doi.org/10.1145/321992.321996
4. Engelfriet, J., Vogler, H.: Macro Tree Transducers. Journal of Computer and System Sciences **31**(1), 71–146 (1985). https://doi.org/10.1016/0022-0000(85)90066-2
5. Engelfriet, J., Vogler, H.: Modular Tree Transducers. Theoretical Computer Science **78**(2), 267–303 (1991). https://doi.org/10.1016/0304-3975(91)90353-4
6. Farzan, A., Nicolet, V.: Counterexample-guided Partial Bounding for Recursive Function Synthesis. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, vol. 12759, pp. 832–855. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_39, series Title: Lecture Notes in Computer Science
7. Gill, A.J.: Cheap deforestation for non-strict functional languages. Ph.D. thesis, University of Glasgow (1996)
8. Hong, Q., Aiken, A.: Recursive Program Synthesis using Paramorphisms. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656381
9. Hu, Z., Iwasaki, H., Takeichi, M.: Calculating Accumulations. New Gener. Comput. **17**(2), 153–173 (1999). https://doi.org/10.1007/BF03037434
10. Hughes, R.J.M.: A Novel Representation of Lists and its Application to the Function "reverse". Inf. Process. Lett. **22**(3), 141–144 (1986). https://doi.org/10.1016/0020-0190(86)90059-1
11. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic Program Synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 944–959. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454087
12. Kühnemann, A., Glück, R., Kakehi, K.: Relating Accumulative and Non-accumulative Functional Programs. In: Goos, G., Hartmanis, J., Van Leeuwen, J., Middeldorp, A. (eds.) Rewriting Techniques and Applications, vol. 2051, pp. 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-45127-7_13, series Title: Lecture Notes in Computer Science

13. Lee, W., Cho, H.: Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). https://doi.org/10.1145/3571263
14. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program Sketching with Live Bidirectional Evaluation. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (Aug 2020). https://doi.org/10.1145/3408991, arXiv:1911.00583 [cs]
15. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (Jan 2022). https://doi.org/10.1145/3498682
16. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 619–630. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2738007
17. Pettorossi, A., Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs. ACM Comput. Surv. **28**(2), 360–414 (1996). https://doi.org/10.1145/234528.234529
18. Voigtländer, J.: Concatenate, reverse and map vanish for free. In: Wand, M., Jones, S.L.P. (eds.) Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. pp. 14–25. ACM (2002). https://doi.org/10.1145/581478.581481
19. Wadler, P.: The Concatenate Vanishes. Dept of Computer Science, Glasgow University (1987)
20. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theor. Comput. Sci. **73**(2), 231–248 (1990). https://doi.org/10.1016/0304-3975(90)90147-A
21. Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided Inductive Synthesis of Recursive Functional Programs. Proceedings of the ACM on Programming Languages **7**(PLDI), 860–883 (Jun 2023). https://doi.org/10.1145/3591255