Recursive Program Synthesis using Paramorphisms

QIANTAN HONG, Stanford University, USA
ALEX AIKEN, Stanford University, USA

We show that synthesizing recursive functional programs using a class of primitive recursive combinators is both simpler and solves more benchmarks from the literature than previously proposed approaches. Our method synthesizes paramorphisms, a class of programs that includes the most common recursive programming patterns on algebraic data types. The crux of our approach is to split the synthesis problem into two parts: a multi-hole template that fixes the recursive structure, and a search for non-recursive program fragments to fill the template holes.

CCS Concepts: • Software and its engineering → General programming languages; Programming by example; Search-based software engineering; Automatic programming.

Additional Key Words and Phrases: Program Synthesis, Examples, Stochastic Synthesis, Recursion Schemes

ACM Reference Format:

1 INTRODUCTION

We consider the problem of synthesizing recursive programs from input-output examples. Following previous work, we consider functional programs over algebraic data types such as the natural numbers, lists, and trees [Kneuss et al. 2013; Lubin et al. 2020; Osera and Zdancewic 2015]. For example, consider a program that appends two lists:

\[
\begin{align*}
\text{append } \text{Nil} & \quad = \quad \text{Nil} \\
\text{append } (\text{Cons } h \ t) & \quad = \quad \text{Cons } h (\text{append } t \ l)
\end{align*}
\]

This program uses general recursion, that is, the function append is explicitly recursively defined, with calls to append within its definition. Depending on what other language features are present, unrestricted general recursion is difficult to reason about; for example, proving termination of general recursive programs is normally non-trivial.

In practice many iterative/recursive programs, including append, can be expressed using more restricted primitive recursive constructs. The essence of primitive recursion is that the number of iterations or recursive invocations is known when the function is first called. For example, the fold combinator captures a typical primitive recursive pattern where the number of recursive calls is the length of the list argument. A standard (general recursive) definition of fold is:

\[
\begin{align*}
\text{fold } \text{Nil} & \quad n \ f \ = \quad n \\
\text{fold } (\text{Cons } h \ t) & \quad n \ f \ = \quad f \ h (\text{fold } t \ n \ f)
\end{align*}
\]

We can use fold to write a well-known alternative definition of append:

\[
\begin{align*}
\text{append } l_1 \ l_2 & \quad = \quad \text{fold } l_2 \ l_1 \ \text{Cons}
\end{align*}
\]
Consider the two definitions of append above. From the point of view of program synthesis, the version using fold has obvious advantages: There is no need to discover where to place recursive calls to "tie the knot", and termination is trivially guaranteed. The main disadvantage of the fold version is that it appears quite restrictive: fold only works for lists, and there are list programs for which a single fold is insufficient.

Our main contribution is a program synthesis algorithm based on paramorphisms [Meijer et al. 1991]. Paramorphisms generalize the fold operator to more recursion patterns as well as other algebraic data types. The paramorphism combinator on lists is:

\[
\text{para } \text{Nil } g_{\text{Nil}} g_{\text{Cons}} = g_{\text{Nil}} \\
\text{para } (\text{Cons } h t) g_{\text{Nil}} g_{\text{Cons}} = g_{\text{Cons}} h (t, \text{ para } t g_{\text{Nil}} g_{\text{Cons}})
\]

Compare the definition of fold to the definition of para in the Cons case, where fold applies the function \(f\) to the head of the list \(h\) and the result of the recursive call on \(t\) and para applies \(f\) to the head \(h\), the tail \(t\), and the result of the recursive call on \(t\). Thus the para combinator gives the function \(f\) more information (all of the original list argument) as well as the recursively processed tail. (The seemingly unnecessary formation of a pair of \(t\) and the result of the recursive call is convenient for the generalization to other data types presented in Section 2.) Here is the append function written with para for lists:

\[
\text{append } l_1 l_2 = \text{para } l_1 g_{\text{Nil}} g_{\text{Cons}} \text{ where } \\
g_{\text{Nil}} = l_2 \\
g_{\text{Cons}} h (t,s) = \text{Cons } h s
\]

Our synthesis algorithm produces programs with one or more uses of para; this last definition of append is one synthesized by our implementation. Our algorithm splits the synthesis problem into two parts:

1. We select a template, which is an incomplete program using para with multiple holes.
2. We synthesize non-recursive program fragments to fill the holes.

The motivation for (1) is that, in practice, there are relatively few programming patterns that account for most forms of recursion and looping, and so with a relatively small number of primitive recursive templates we can capture a large fraction of natural loops and recursive functions. The template that leads to successful synthesis of the append function is

\[
\text{append } l_1: \text{ListNat } l_2: \text{ListNat } = \\
\text{para } [ ]_1 \text{ListNat } g_{\text{Nil}} g_{\text{Cons}} \text{ where } \\
g_{\text{Nil}} = [ ]_1 \text{ListNat} \\
g_{\text{Cons}} h (t,s) = [ ]_2 \text{ListNat} \\
\]

Note that templates are typed, and in particular the holes can only be filled with terms of a particular type. Thus, for each template, we also consider multiple possible variations where the holes are assumed to have different specific types. We discuss templates in detail in Section 3.2.

The motivation for (2) is that once the recursive structure is fixed, the synthesis problem reduces to the simpler problem of synthesizing non-recursive code. For (2) we use stochastic search techniques inspired by [Schkufza et al. 2013]. In this approach, an initial program (which is the chosen template filled with some default program fragments) is incrementally changed with randomly chosen modifications guided by a cost function. The cost function evaluates how "close" the program is to satisfying all of the input-output pairs that form the specification of the desired program; programs that have fewer differences in the required output have better scores than programs that produce more differences. In [Schkufza et al. 2013] the cost is based on the number of bits of output...
that differ. Because we work on algebraic data types that have more structure than collections of machine words, we require a different measure of cost; see Section 3.1.2.

We have implemented our synthesis algorithm in a tool PARA, which we have applied to 59 benchmarks, including the benchmarks of [Lubin et al. 2020] collected from previous papers on synthesizing recursive programs and a number of new benchmarks we have written. In our experiments PARA solves 55/59 benchmarks, while SMYTH [Lubin et al. 2020], $\lambda^2$ and TRIO [Lee and Cho 2023], three state-of-the-art recursive synthesis systems, solve 33/59, 31/54 and 41/59 benchmarks respectively (five benchmarks require higher-order input, which $\lambda^2$ does not support). All systems run in similar time for the problems they successfully solve, requiring fractions of a second to less than a minute on conventional hardware. See Section 5 for a more in-depth discussion of the experiments. We note that other systems are designed to handle synthesis of general recursive functions. However, none of the benchmark programs that appear in previous papers actually require general recursion—all previously reported results are on programs that are in fact primitive recursive. The fact that PARA is able to solve almost all benchmarks even though those benchmarks were not originally intended to be primitive recursive shows that appropriately chosen primitive recursive forms are quite expressive.

To summarize, we make the following contributions:

- We develop an approach to synthesis of primitive recursive programs over algebraic data types based on paramorphisms.
- We decompose the synthesis problem into a set of multi-hole paramorphic templates and a stochastic search to fill the holes with non-recursive program fragments.
- We show that this simple algorithm, with appropriately chosen templates, is able to solve all of the benchmarks from previous work plus a number of new and more challenging benchmarks.

The rest of the paper is organized as follows. Section 2 introduces the simply-typed functional language we use as the target of synthesis. Section 3 defines templates and our synthesis algorithm. Section 4 describes the implementation of PARA, Section 5 presents the experimental results, Section 6 discusses related work, and Section 7 concludes.

2 PROGRAMMING LANGUAGE

Our target language PARA, shown in Figure 1, is a simply-typed functional language with paramorphisms. A PARA program consists of a list of type declarations `typedecs` and a program term `t`. Type declarations in PARA are standard, potentially recursive, algebraic data types [Milner et al. 1997]. We will refer to the $k$-th constructor of a type $T$ as $Ctr_k^T$.

Without loss of generality, we require that the declaration of a type $T$ (the first line of Figure 1) list all non-recursive arguments of type constructors (i.e., constructor arguments of types other than
$T$) before any recursive arguments (i.e., constructor arguments of type $T$). This ordering simplifies notation in our algorithms for generating templates in Section 3.2. For example, the definitions of the natural numbers and the booleans

```haskell
data Nat = Zero | Succ Nat
data Bool = False | True
```

have only constructors that take zero or one type arguments, so the ordering requirement imposes no constraint. Now consider lists of natural numbers and binary trees that store natural numbers at interior nodes:

```haskell
data ListNat = Nil | Cons Nat ListNat
data TreeNat = Leaf | Node Nat TreeNat TreeNat
```

In the definition of the Cons constructor of ListNat the argument of type ListNat is last. Similarly, in the Node constructor of TreeNat the two recursive constructor arguments of type TreeNat are also listed last. We use the notation $Ctr^T_1 T_1 \cdots T \cdots$ for pattern matching, where $T_1 \cdots$ should be understood to match the non-recursive constructor arguments of $Ctr^T_k$ and $T \cdots$ matches the recursive constructor arguments. For simplicity we consider only monomorphic algebraic data types. Our implementation supports polymorphic type declarations by instantiating them with monomorphic types during type checking.

The standard terms of the language are variables, $\lambda$-abstractions, function applications and applications of constructors. para terms support primitive recursive computation via paramorphisms; for a detailed review of paramorphisms from a formal perspective, see [Meertens 1992]. Section 1 gives an overview of paramorphisms and an example list computation. Figure 1 generalizes this example to an arbitrary algebraic datatype $T$. The para combinator uses the locally defined functions $g_1 \cdots g_m$ in a case analysis on the base functor datatype of $T$. Each case takes a number of arguments equal to the arity of the constructor; the first arguments (of types other than $T$) are the corresponding non-recursive constructor arguments, and the remaining arguments (corresponding to constructor arguments of type $T$) are pairs consisting of the original recursive constructor arguments and the result of a recursive call. Using pairs here allows us to have a single value corresponding to each constructor argument even though paramorphisms treat recursive constructor arguments of type $T$ itself differently from other types. The monomorphic typing rules for Para are straightforward and given in Figure 2.

The evaluation rules for Para are given in Figure 3. The $\beta$-reduction of $\lambda$-terms is standard. The rule for para terms applies when the first argument to para is a constructed term with constructor $Ctr^T_k$. The reduction is carried out by selecting the corresponding $k$th function supplied to para:

$$g_k u_{k1} \cdots (u_{kl}, u'_{kl}) \cdots = t'$$

then substituting each actual constructor argument $t_i$ for $u_{ki}$ in $t'$. Furthermore, if the $i$th constructor argument is recursive (i.e., of type $T$), the result of a recursive call to para on argument $t_i$ is substituted for $u'_{ki}$.

We conclude this section with two additional programs written in Para. The first example uses a provided library multiplication function mul of type $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ to define factorial. Factorial is an example of a primitive recursive function that uses the expressiveness of paramorphisms—note that in the Succ case both the original constructor argument pred and the recursively computed value state are used to compute the result:

```haskell
factorial = \_n::Nat.
  para n gZero gSucc where
  gZero = 1
```

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 151. Publication date: June 2024.
Recursive Program Synthesis using Paramorphisms

Factorial is one of our synthesis benchmarks. In our experiments, however, we synthesize the entire program, including multiplication and its required building block, addition, from scratch; see Section 5.

The second example maps a function over a tree of natural numbers. Note that this program does not use the full power of paramorphisms; the subtree arguments in the Node case are not needed to rebuild the tree:

\[
\text{tree-map} = \lambda f : \text{Nat} \rightarrow \text{Nat}. \lambda \text{tree} : \text{TreeNat}. \\
\text{para tree } g_{\text{Leaf}} g_{\text{Node}} \text{ where} \\
\qquad g_{\text{Leaf}} = \text{Leaf} \\
\qquad g_{\text{Node}} n (\text{subtree}_1, \text{state}_1) (\text{subtree}_2, \text{state}_2) = \text{Node } (f n) \text{ state}_1 \text{ state}_2
\]

\[g_{\text{succ}} \ (\text{pred}, \text{state}) = \text{mul } (\text{Succ } \text{pred}) \text{ state}\]

\[\text{tree-map} = \lambda f : \text{Nat} \rightarrow \text{Nat}. \lambda \text{tree} : \text{TreeNat}. \\
\text{para tree } g_{\text{Leaf}} g_{\text{Node}} \text{ where} \\
\qquad g_{\text{Leaf}} = \text{Leaf} \\
\qquad g_{\text{Node}} n (\text{subtree}_1, \text{state}_1) (\text{subtree}_2, \text{state}_2) = \text{Node } (f n) \text{ state}_1 \text{ state}_2
\]

\[\text{tree-map} = \lambda f : \text{Nat} \rightarrow \text{Nat}. \lambda \text{tree} : \text{TreeNat}. \\
\text{para tree } g_{\text{Leaf}} g_{\text{Node}} \text{ where} \\
\qquad g_{\text{Leaf}} = \text{Leaf} \\
\qquad g_{\text{Node}} n (\text{subtree}_1, \text{state}_1) (\text{subtree}_2, \text{state}_2) = \text{Node } (f n) \text{ state}_1 \text{ state}_2
\]

\[g_{\text{succ}} \ (\text{pred}, \text{state}) = \text{mul } (\text{Succ } \text{pred}) \text{ state}\]

3 SYNTHESIS ALGORITHM

Given a set of input-output examples, our goal is to synthesize a PARA program that returns the correct output for every input in the example set.

\[g_{\text{succ}} \ (\text{pred}, \text{state}) = \text{mul } (\text{Succ } \text{pred}) \text{ state}\]
\[
\begin{align*}
    u &= v \mid Ctr^{T} u_1 \cdots u_n \\
    s &= [ ]^T_I \mid \lambda v : \tau . s \\
    \mid \text{para } t^k g_1 \cdots g_n \text{ where} \\
    & \quad g_1 v \cdots (v, v) \cdots = s \\
    & \quad \cdots \\
    & \quad g_m v \cdots (v, v) \cdots = s
\end{align*}
\]

Fig. 4. Stratified grammar PARA\(^d\) and PARA\(^s\)

Problem 3.1. Given a set of \(N\) input-output examples \(\{(I_i, O_i)\}\), find the smallest PARA term \(t\) satisfying all examples, i.e. \(\land_{i=0}^{N} t I_i \to O_i\) is true, where \(\to\) is the reflexive transitive closure of the reduction relation in Figure 3.

We stratify PARA into a non-recursive fragment PARA\(^d\) and a template fragment PARA\(^s\). Non-recursive terms are a syntactic subset of PARA terms, and templates are multi-hole contexts that yield full PARA terms when all holes are substituted with non-recursive terms. We denote the result of substituting a sequence of non-recursive terms \(\bar{u}\) into corresponding holes in a template \(s\) by \(\bar{s}\).

The stratified grammar only generates a subset of the full PARA grammar, as it disallows occurrences of PARA\(^s\) terms inside constructor terms. Nevertheless, it still preserves the full expressiveness of PARA. Note that a PARA program that has PARA\(^s\) terms inside constructor terms can always be rewritten into one without such occurrences by hoisting the PARA\(^s\) terms out of the constructor term and introducing a variable, e.g.

\[
\text{Cons } (\text{para } \cdots \text{ where } \cdots ) y = (\lambda z . \text{Cons } z y) (\text{para } \cdots \text{ where } \cdots )
\]

By stratifying PARA’s grammar, we can decompose synthesis problems into two subproblems:

1. Find a template, such that some sequence of non-recursive terms solves the synthesis problem when substituted into the template;
2. Given a template, find a sequence of non-recursive terms that solves the synthesis problem when substituted into the template, if possible.

The intuition behind the stratification of the grammar is that relatively few recursion patterns account for most of what is written in practical programming, and thus most programs of interest should be expressible by a relatively small set of templates. For algebraic data types in particular, paramorphisms capture common patterns of recursion and allow us to focus the synthesis procedure on the easier problem of finding non-recursive terms to fill the holes.

Our synthesis algorithm combines an outer loop that enumerates an expressive but relatively small set of templates and an inner loop that solves for non-recursive terms given a fixed template. The inner loop problem is simpler than searching for a whole program directly because it need not invent recursion patterns. The synthesis algorithm does not terminate until a solution is found or the total time budget is exceeded. As mentioned in Section 1, we address the problem of finding the non-recursive terms by leveraging stochastic program synthesis techniques [Schkufza et al. 2013]. We outline the synthesis algorithm in Figure 5. We write \text{normalize}(t)\ for the normalization of a term \(t\) under \(\to\). The functions stochastic-synthesis and template-set are discussed in Sections 3.1 and 3.2, respectively.
Input: Algebraic data type declarations \( \vec{d} \), library function declarations \( \Gamma \), expected program type signature \( \tau_i \rightarrow \tau_o \), input-output examples \( \{(I_i, O_i)\} \)

Output: A PARA program \( t \) of type \( \tau_i \rightarrow \tau_o \) that satisfies all input-output examples

\[ S \leftarrow \text{template-set} \left( \vec{d}, \Gamma, \tau_i \rightarrow \tau_o \right) \]

while true

\[ \vec{a}, \text{success} \leftarrow \text{stochastic-synthesis}(s, \{(I_i, O_i)\}) \]

if success then

\[ \text{return normalize}(s[\vec{a}]) \]

Fig. 5. Outline of the synthesis algorithm

3.1 Synthesizing Non-Recursive Terms

3.1.1 Rewriting Terms. Given a template \( s \), we initialize each hole of type \( T \) in the template with the default candidate of its type, denoted by \( D(T) \):

\[
\begin{align*}
\text{(data } T & = \cdots \mid Ctr^T_{\text{min}} T_1 \cdots \mid \cdots \text{) } \in \text{typedcls} \quad \frac{\text{D}(T) = Ctr^T_{\text{min}} D(T_1) \cdots}}{}
\end{align*}
\]

where \( Ctr^T_{\text{min}} \) is a non-recursive constructor of \( T \) with the smallest arity. When multiple such constructors exist, we choose one arbitrarily.

Our synthesizer searches for non-recursive terms by beginning with a default candidate and randomly applying a set of rewrite rules \( \rightarrow^* \) parameterized by the typing context \( \Gamma \) for the hole:

\[
\begin{align*}
\text{(data } T & = \cdots \mid Ctr^T_k T_1 \cdots T \cdots \mid \cdots \text{) } \in \text{typedcls} \quad \frac{\text{\Gamma } \vdash u : T}{u \rightarrow^\Gamma Ctr^T_k D(T_1) \cdots u \cdots} \quad \text{(CONSTRUCTION)}
\end{align*}
\]

\[
\begin{align*}
\text{(data } T & = \cdots \mid Ctr^T_k T_1 \cdots T \cdots \mid \cdots \text{) } \in \text{typedcls} \quad \frac{\text{\Gamma } \vdash t : T, v : T}{\text{\Gamma } \vdash t \rightarrow^\Gamma v} \quad \text{(VARIABLE)}
\end{align*}
\]

The CONSTRUCTION rule creates a new constructor application (by using the term \( u \) in a fresh constructor of the same type where all other positions of the constructor are filled with default terms), the PROJECTION rule replaces a constructor application with one of its (same-typed) arguments, and VARIABLE replaces a term with a variable of the same type. We omit subscript \( \Gamma \) when it is clear from context. We can reach any well-typed non-recursive term starting from an arbitrary well-typed non-recursive term by rewriting under \( \rightarrow^\Gamma \):

**Theorem 3.2.** If \( \Gamma \vdash u_1, u_2 : T \), then there exists a rewrite sequence \( r \in \rightarrow^* \) such that \( r(u_1) = u_2 \).

**Proof.** First, observe that one step of CONSTRUCTION rewrites any well-typed term to \( D(T) \) by substituting \( Ctr^T_k \) with \( Ctr^T_{\text{min}} \) (this is a degenerate application of CONSTRUCTION where \( u \) does not appear on right-hand side, because \( Ctr^T_{\text{min}} \) must be non-recursive). This gives a rewrite sequence \( r_1 \) such that \( r_1(u_1) = D(T) \). We then define a rewrite sequence \( r_2 \) such that \( r_2(D(T)) = u_2 \) by induction on a well-typed derivation of \( u_2 \).

1) If \( u_2 \) is a variable, then one step of VARIABLE rewrites any term to \( u_2 \). 2) If \( u_2 \) is a constructor application of \( Ctr^T_k \), for each argument \( u'_i \) we define a rewriting sequence \( r'_i \) such that \( r'_i(D(T_i)) = u'_i \) by induction. Then, one step of CONSTRUCTION rewrites \( D(T) \) to \( Ctr^T_k D(T_1) \cdots \), and we then apply each rewriting sequence \( r'_i \) to each argument.
sub-term, which rewrites $\text{Ctrl}_k^T D(T_1) \cdots$ to $\text{Ctrl}_k^T u'_1 \cdots = u_2$. The concatenation of $r_1$ and $r_2$ is the desired sequence $r$.

Thus, if some sequence of non-recursive terms $u_1 \cdots u_n$ solves a synthesis problem by substituting that sequence into the holes of the template $s$, then it is always possible to obtain that solution by applying some $\rightsquigarrow$ rewrite sequence to the default candidates $D(T_1) \cdots D(T_n)$.

3.1.2 Cost Function. Naive breadth-first search of rewrite sequences gives us a semi-decision procedure for deciding whether there exists a sequence of non-recursive terms that solves a synthesis problem. However, as we show in our experiments, searching for a solution by enumerating all such possible rewrites is very expensive. Instead, we use a stochastic search procedure where the synthesis problem is recast as a cost minimization problem, and we apply $\rightsquigarrow$ rewrites stochastically guided by the cost function.

Our cost function $\text{Cost}$ consists of two terms: $\text{Cost}_{\text{size}}$ penalizes program size, and $\text{Cost}_{\text{error}}$ penalizes incorrect outputs. A constant $\alpha$ adjusts the weight of the two terms:

$$\text{Cost}(\overline{u}) = \text{Cost}_{\text{size}}(\overline{u}) + \alpha \cdot \text{Cost}_{\text{error}} = \sum_{u \in \overline{u}} \log \text{size}(u) + \alpha \cdot \sum_{i=0}^{N} \log \text{error}(\text{normalize}(s[\overline{u}] I_i), O_i)$$

The cost minimization procedure minimizes $\text{Cost}$ and reports any candidate sequence of non-recursive terms $\overline{u}$ encountered during the search such that $\text{Cost}_{\text{error}}(\overline{u}) = 0$. Given a large enough $\alpha$, solutions to Problem 3.1, if any exist within the subset of programs defined by the template, are global minima of $\text{Cost}$. In practice, setting $\alpha = 1$ usually allows us to visit good candidates with correct behavior and small sizes when searching near minima of $\text{Cost}$.

Both size($\cdot$) and error($\cdot$, $\cdot$) are defined as the minimal length of rewrite sequences satisfying some property. For size($u$), the rewrite sequence must rewrite the default candidate term $D(T)$ into current candidate term $u$. For error($I$, $O$), the rewrite sequence must rewrite the program output $O' = \text{normalize}(s[\overline{u}] I_i)$ to the desired output $O$:

$$\text{size}(u) = \min_{r \in \rightsquigarrow_r^T} \text{length}(r) \text{ s.t. } s(D(T)) = u$$

$$\text{error}(O', O) = \min_{r \in \rightsquigarrow_r^T} \text{length}(r) \text{ s.t. } s(O') = O$$

For our choice of $D(T)$, the exact value of size($\cdot$) is computed by the following recursive procedure

$$\text{size}(u) = \begin{cases} 1 & \text{if } u = \epsilon \\ \sum_{i=1}^{n} \text{size}(u_i) & \text{if } u = u_1 \cdots u_n \end{cases}$$

$$\text{size}(\text{Ctrl}_k^T u_1 \cdots u_n) = \begin{cases} \text{size}(u_1) & \text{if } \text{Ctrl}_k^T \neq \text{Ctrl}_k^T_{\text{min}} \\ \sum_{i=1}^{n} \text{size}(u_i) & \text{otherwise} \end{cases}$$

The exact value of error($\cdot$, $\cdot$) is difficult to compute efficiently, so we approximate it using a tree edit-distance algorithm in our implementation.

To minimize $\text{Cost}$, we employ a stochastic Monte Carlo search procedure inspired by Metropolis–Hastings sampling [Hastings 1970]. Our sampling procedure works as follows: at each step we maintain a current candidate $\overline{u}$ and its cost $\text{Cost}(\overline{u})$ and create a modified candidate $\overline{u}'$, called the proposal, by uniformly sampling and applying one $\rightsquigarrow$ rewrite among all possible $\rightsquigarrow$ rewrites to any non-recursive term in $\overline{u}$. We then compute $\text{Cost}(\overline{u}')$. The proposal is accepted, meaning it becomes the new current candidate, with the following probability, otherwise the current candidate is retained to the next step:

$$A(\overline{u}', \overline{u}) = \min \left( 1, \exp(-\beta(\text{Cost}(\overline{u}') - \text{Cost}(\overline{u}))) \right)$$

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 151. Publication date: June 2024.
where $\beta$ is a constant. The proposal is always accepted if its cost is lower than the current candidate, otherwise the proposal is accepted with a probability that depends on the difference in cost between the proposal and the current candidate, with more expensive proposals having lower probability of acceptance. We repeat this procedure, proposing and possibly accepting one new candidate per step, until a solution is found or the computation budget is exhausted.

When the sampling procedure converges, more samples will be taken for which $\text{Cost}(v) < \text{Cost}(u)$, giving us high probability of finding the global minima or sufficiently good candidates. Even before the sampling converges, this approach effectively hill climbs to nearby locally good solutions, but always has some probability of jumping to other parts of the search space.

3.1.3 Example. To illustrate how stochastic rewrites guided by the cost function synthesize non-recursive terms, we show several steps of an execution trace of the Para synthesizer synthesizing an append function for ListNat.

At the beginning of the search, each hole is initialized with its default candidate. In this example, every hole has type ListNat and is initialized to Nil:

candidate$_A = \lambda l_1 : \text{ListNat} \, \lambda l_2 : \text{ListNat}.
\begin{align*}
\text{para } [\text{Nil}]_1^\text{ListNat} & \, \text{gNil} \, \text{gCons } \text{where} \\
g\text{Nil} & = [\text{Nil}]_2^\text{ListNat} \\
g\text{Cons} \, h \, (t, s) & = [\text{Nil}]_3^\text{ListNat}
\end{align*}

This initial function always returns Nil, regardless of the input. Figure 6 plots the cost of a search beginning with this program, labelled A in the figure.

The first significant decrease in the cost function occurs after fewer than ten accepted proposals when the term in the second hole is rewritten from Nil to $l_2$, which can be done by one step of Variable:

candidate$_B = \lambda l_1 : \text{ListNat} \, \lambda l_2 : \text{ListNat}.
\begin{align*}
\text{para } [\text{Nil}]_1^\text{ListNat} & \, \text{gNil} \, \text{gCons } \text{where} \\
g\text{Nil} & = [l_2]_2^\text{ListNat} \\
g\text{Cons} \, h \, (t, s) & = [\text{Nil}]_3^\text{ListNat}
\end{align*}

We now have a function that always returns its second list argument. This function has lower cost because the second list is more similar to the desired output (the concatenation of the two lists). The search discovers this function at point B in Figure 6.

The next significantly better function is not found until point C, after the 50th accepted proposal and close to the end of the search. This program iterates over $l_1$ and adds a constant 0 to the head of the result list for every iteration:

candidate$_C = \lambda l_1 : \text{ListNat} \, \lambda l_2 : \text{ListNat}.
\begin{align*}
\text{para } [l_1]_1^\text{ListNat} & \, \text{gNil} \, \text{gCons } \text{where} \\
g\text{Nil} & = [l_2]_2^\text{ListNat} \\
g\text{Cons} \, h \, (t, s) & = [\text{Cons 0 s}]_3^\text{ListNat}
\end{align*}

The function has lower cost because its output is even closer to the desired output, with the correct length but potentially incorrect elements in the final list. The term in the first hole is rewritten from Nil to $l_1$, which is achievable via one step of Variable. The term in the second hole has not changed. The term in the third hole is rewritten from Nil to Cons 0 s, which can be achieved by first rewriting to s via one step of Variable and then to Cons 0 s via one step of Construction.

Finally, one step of Variable rewrites the subterm 0 inside Cons 0 s to h which results in a correct append function, discovered at point D in Figure 6:
append = \lambda l_1 : \text{ListNat} \ \lambda l_2 : \text{ListNat}. \\
\text{para } [l_1]_{\text{ListNat}} g_{\text{Nil}} g_{\text{Cons}} \text{ where } \\
g_{\text{Nil}} = [l_2]_{\text{ListNat}} \\
g_{\text{Cons}} \ h \ (t, s) = [\text{Cons } h \ s]_{\text{ListNat}}

We note that while only the \text{Variable} and \text{Construction} rewrites are strictly necessary for this example, the \text{Projection} rule is important in practice because it allows the search to undo decisions that end up leading nowhere. While there is a very short path from the initial program to a correct solution, the actual search trajectory explores many more programs, which is typical behavior for stochastic search. The long plateau in Figure 6 occurs because multiple changes must be present simultaneously for the cost to improve significantly from point B to point C, and even with a good cost function many trials are needed to discover the right combination of modifications.

3.2 Enumerating Templates
3.2.1 Proper Recursion Nests. In this paper we focus on templates representing a single proper recursion nest of one or more calls to \text{para}, which we will show is expressive enough to solve our benchmarks. It is possible to extend the set of templates to include several sequentially chained proper recursion nests, but we do not consider this possibility in this paper.

We define a template generator \text{proper-nest} in Figure 7. Because \text{proper-nest} is a code generator, its function body is a mix of code to be executed as part of \text{proper-nest} itself and code that is the output of \text{proper-nest}. We color-code what is part of the output of \text{proper-nest} in blue; code in black is part of \text{proper-nest}’s own logic.

All variables in the generated code are implicitly renamed to distinct fresh variables unless explicitly labeled by subscripts. All holes are implicitly given distinct labels. The function \text{proper-nest} generates a template term given a state type and a recursion type list. The state type is the return type of all \text{para} terms in the template. The recursion type list specifies the maximal depth of the recursion nest and the types on which \text{para} terms recurse: A \text{para} term at recursion nesting depth \( n \) consumes a structure with the type specified by the \( n \)-th element of the recursion type list. We use Haskell syntax \((\text{head} :: \text{tail})\) and \([\ ]\) to pattern match on the meta-level recursion type list, to distinguish it from object-level lists denoted by constructor syntax \text{Cons} \text{head} \text{tail} and \text{Nil}.

When we write patterns of the form \( g_j v \cdots (v, v) \cdots \), the \( v \cdots \) should be understood to match
the non-recursive arguments to \( g_j \) and \((v, v) \cdots \) matches the recursive arguments, similar to our
convention of writing non-recursive constructor arguments before recursive constructor arguments
introduced in Section 2.

There are two cases for proper-nest, depending on state type argument \( S \). The case for when
\( S \) is an algebraic data type is defined in Figure 7a. For \( \text{para} \) terms consuming a recursive type,
proper-nest only generates \( \text{para} \) subterms for cases where the constructor has some recursive
arguments. For \( \text{para} \) terms that consume non-recursive types (which degenerates into a pattern
match), we generate \( \text{para} \) subterms for all cases.

The template for append given in Section 1 is generated by the call proper-nest \( \text{ListNat} \}
\([\text{ListNat}]\}. As a more involved example, proper-nest \( \text{ListNat} \} \{\text{Boolean, TreeNat}\} \}
generates the following template:

\[
\begin{align*}
\text{para} & [ \frac{1}{1} \text{Boolean} g_{\text{true}} g_{\text{false}} \text{ where } \\
& g_{\text{true}} = \text{para} [ \frac{1}{2} \text{TreeNat} \text{, } g_{\text{leaf}} g_{\text{node}} \text{ where } \\
& g_{\text{leaf}} = [ ]_{3} \text{ListNat} \\
& g_{\text{node}} n (\text{subtree}_1, \text{state}_1) (\text{subtree}_2, \text{state}_2) = [ ]_{4} \text{ListNat} \\
& g_{\text{false}} = \text{para} [ \frac{1}{2} \text{TreeNat} \text{, } g_{\text{leaf}} g_{\text{node}} \text{ where } \\
& g_{\text{leaf}} = [ ]_{5} \text{ListNat} \\
& g_{\text{node}} n (\text{subtree}_3, \text{state}_3) (\text{subtree}_4, \text{state}_4) = [ ]_{7} \text{ListNat}
\end{align*}
\]

It turns out that it is useful to allow state types to be not just algebraic data types but also
functions. Function types can encode top-down recursion via continuation-passing style in addition
to the bottom-up recursion expressed “natively” by paramorphisms. For example, one way to reverse
a list is to traverse the list from head to tail and add the visited element to the head of the result list
during traversal. This algorithm can be expressed with \( \text{para} \) by using \( \text{NatList} \rightarrow \text{NatList} \) as
the state type, even though \( \text{para} \) traverses the list from tail to head:

\[
\begin{align*}
\text{reverse} & = \lambda l: \text{ListNat}. \\
& \quad (\text{para } l \ g_{\text{nil}} g_{\text{cons}} \text{ where } \\
& \quad g_{\text{nil}} = \lambda l_1: \text{NatList}. l_1 \\
& \quad g_{\text{cons}} h (t, s) = \lambda l_2: \text{NatList}. (\text{Cons } h l_2)) \text{ Nil}
\end{align*}
\]
Figure 7b defines how proper-nest handles function types as state types using an auxiliary function proper-nest-func. The definition of proper-nest-func is similar to the algebraic data type case of proper-nest. The base case is adjusted to keep the generated term syntactically well-formed. We wrap a pair of a λ-abstraction and an application around the term generated by proper-nest-func so that the input to the state (which is a function) can be varied.

3.2.2 Library Functions and Tuples. It can be useful to supply a set of library functions to be used as additional primitives in the synthesis of non-recursive terms. We support first-order library functions by amending production rules to the grammar of non-recursive terms and adding the corresponding typing rules and rewrite rules, similar to the approach described in [Alur et al. 2013]. These terms representing library function applications can then be used in any holes in the template, including the data to be pattern-matched by the para combinators. For example, to include the library function +: Nat × Nat → Nat, we add a new production:

\[ u = \cdots | u + u \]

and the corresponding typing rule:

\[ \Gamma \vdash u_1 : \text{Nat}, u_2 : \text{Nat} \]
\[ \Gamma \vdash u_1 + u_2 : \text{Nat} \]

We add the following rewrite rules so that the search procedure can construct and remove non-recursive terms representing addition:

\[ u \leadsto u + u \]
\[ u_1 + u_2 \leadsto u_1 \]
\[ u_1 + u_2 \leadsto u_2 \]

As another example, for the library function =: Nat × Nat → Bool, the following rewrite rule is added:

\[ \Gamma \vdash u : \text{Bool} \]
\[ u \leadsto D(\text{Nat}) = D(\text{Nat}) \]

where the default candidate \( D(\text{Nat}) = \text{Zero} \). Note that because the return type Bool does not occur in the argument types of the function =, there is no dedicated rewrite rule to remove terms of the form \( u = u \) while retaining some subterm(s). Instead, the search procedure can remove such terms by invoking the CONSTRUCTION rule, which rewrites any of them to the default candidate \( D(\text{Bool}) = \text{False} \).

We also support using tuple types as state types. Tuple types can encode recursion over multiple state components by packaging them as a tuple. For a sequence of types \( \tau_1 \cdots \tau_n \), we generate the type declaration of their tuple type if used, denoted by \( \tau_1 \times \cdots \times \tau_n \), as follows:

\[ \text{data } \tau_1 \times \cdots \times \tau_n = \text{Tuple } \tau_1 \cdots \tau_n \]

We then introduce the projections \( \pi_1 \cdots \pi_n \) for this tuple type as library functions. Projection \( \pi_i \) is:

\[ \pi_i (\text{Tuple } t_1 \cdots t_i \cdots t_n) = t_i \]

3.2.3 Template Set. We generate a set of templates by applying the generator proper-nest to different plausible combinations of state type and recursion type list. Currently, we use the following strategy to generate the template set:

\[ \text{template-set} = \{ \text{proper-nest } S [T_1 \cdots T_D] | S, T_1 \cdots T_D \in \{ T_m \} \} \]
\[ \cup \{ \text{proper-nest } S_1 \rightarrow S_2 [T_1 \cdots T_D] | S_1, S_2, T_1 \cdots T_D \in \{ T_m \} \} \]
\[ \cup \{ \text{proper-nest } S_1 \times S_2 [T_1 \cdots T_D] | S_1, S_2, T_1 \cdots T_D \in \{ T_m \} \} \]
where $D$ is a parameter specifying the recursion depth of the templates and $\{T_{in}\}$ is the set of algebraic data types appearing in the type signatures of the target program and library functions.

We then wrap the generated templates appropriately to provide the correct top-level type. For a template $s$ generated with state type $\sigma$ and expected output type $T_1 \to \cdots \to T_n \to T$, we wrap the template inside $\lambda x_1 : \tau_1 \cdots \lambda x_n : \tau_n \cdot ((\lambda x : \sigma \cdot [\ ]^T) \ s)$, where $x_1, \cdots, x_n$ and $x$ are fresh variables.

### 3.3 Normalization

In many cases our synthesizer finds a solution using a template larger than what is strictly necessary. Such solutions typically contain trivial calls to para that do not perform any substantial recursive computation. To eliminate those trivial terms and improve the readability of the programs, we normalize the output of the synthesizer according to the rewriting semantics presented in Figure 3.

### 4 IMPLEMENTATION

We have implemented our synthesis algorithm in a tool also called Para in approximately 700 lines of Common Lisp, excluding the experimental setup. Our prototype supports a predefined set of polymorphic algebraic data types, including List $a$, Tree $a$ and Tuple $a b$ as well as Boolean and Nat. As mentioned previously, before performing synthesis all polymorphic types are resolved to concrete types—synthesis is done on terms with monomorphic types.

**Compiling Templates.** Because each template will likely be executed with many instantiations of non-recursive terms during the synthesis process, it is beneficial to compile a specialized interpreter for each template that executes its candidate non-recursive term sequences. We generate type-annotated Lisp source of a specialized interpreter given a template term and compile it to native code using the SBCL compiler [Rhodes 2008] with high optimization settings (optimize (speed 3)). Compilation results are cached and reused if the same template term is used in multiple problems. We run compilation of different templates in parallel when preparing the template set for a problem.

**Restart Strategy.** We have observed that the distribution of search times for a synthesis problem can be extremely wide. Since there are generally short search paths to a solution, a couple of good guesses early in a search can lead to rapid convergence to a solution, while a couple of bad initial guesses can result in searches taking orders of magnitude longer. As suggested in [Koenig et al. 2021], we periodically restart the search to improve the chances of a short successful search. We use a simple strategy that restarts the search when either of the following conditions are met:

1. After $N_{restart}$ states are explored, but no solution is found.
2. If the wall-clock time for the current search takes a factor of $k_{restart}$ times longer than the average time of all previous searches. The first search is initialized with a timeout of $t_0$.

$N_{restart}$, $k_{restart}$ and $t_0$ are fixed parameters. The second condition also guards against search paths that contain programs of very high time complexity, which can consume significant computation budget while making little progress.

**Parallel Search.** Our synthesis technique is embarrassingly parallel. The search process for different templates can be run simultaneously, and any single template can be accelerated by running multiple stochastic searches in parallel with independent random number generators. Currently, our prototype supports single-node shared-memory parallelism via multi-threading.

### 5 EVALUATION

This section presents experiments designed to answer the following questions:

1. Is Para able to solve difficult synthesis problems with complex recursive patterns?
(2) How does Para’s ability to solve synthesis problems compare with state-of-the-art Program-by-Example systems?

(3) How does Para’s stochastic search procedure compare with enumeration-based search?

(4) How does the ability of state-of-the-art Program-by-Example systems to utilize primitive recursive combinators and automatically-generated templates compare with Para?

Experiments used a 2017 iMac Pro with a 3.2 GHz 8-Core Intel Xeon W CPU and 32 GB of RAM.

5.1 Benchmarks and Setup

We have assembled a suite of 59 synthesis benchmarks for functional programs from previous SMYTH benchmarks [Lubin et al. 2020], the Haskell Prelude library and the Haskell Data.List library [Marlow et al. 2010]. Each benchmark problem is defined by the expected monomorphic type of the top-level program, library functions which the synthesizer may use, and 10 sets of input-output examples, each of which has 16 input-output pairs. The input-output example sets are automatically generated by calling a reference implementation with randomly-sampled inputs. The inputs are generated by reinterpreting algebraic data type definitions as probabilistic context-free grammars [Booth and Thompson 1973], with equal probabilities assigned to each constructor case. We discard any examples that exceed a maximum size (we use 50 nodes as the limit in our experiments). For higher-order problems, inputs of function type are instead uniformly sampled from a set of handwritten functions collected from the SMYTH benchmarks.

All four systems Para, SMYTH, $\lambda^2$ and Trio support supplying library functions, and some of the benchmarks from the literature that we incorporate require library functions to be provided. For these benchmarks, we also include variants where no library functions are provided, which we expect to be a harder problem to solve. We show the list of provided library functions (if any) in parentheses after the name of a benchmark problem.

To answer research questions 1 and 2, we ran Para, SMYTH and Trio on each input-output example set for all 59 benchmarks. We ran $\lambda^2$ on a subset of 54 of our benchmarks that excludes benchmarks with high-order inputs, because $\lambda^2$ does not support them. By default Para uses all 8 cores of the experiment machine and is given a 1-minute time limit for each synthesis task. SMYTH and $\lambda^2$ do not support parallelism and are given an 8-minute computation budget on a single core for each synthesis task. To control for any benefits of the parallel search and enable a more direct comparison to the sequential systems, we also run Para on a single core with the same settings and an 8-minute computation budget, with results labelled by Para (Serial). $\lambda^2$ requires a component library to be provided and we use its standard component library for all runs of $\lambda^2$. This standard library already contains the library functions required by some of our benchmark problems; for these problems, the variants with and without library function become the same for $\lambda^2$, therefore we run $\lambda^2$ on such benchmarks only once and duplicate the resulting entry.

Synthesized programs are validated against an independently generated set of 20 random input-output test cases and then manually inspected for correctness. We report the number of correct synthesis results among 10 runs and the average time of successful runs in seconds. If no synthesis runs are successful, we report the most common reason for failure among the 10 runs: “time” for timeout, “mem” for out of memory, “wrong” for a result with different semantics from the reference implementation, and “fail” for other conditions.

We set parameters $\beta = 5.0, N_{\text{restart}} = 50000, k_{\text{restart}} = 1.5$ and $t_0 = 1s$ for Para. We have verified that Para’s performance is insensitive to small changes of these parameters. We varied the value of the recursion depth parameter $D = 1, 2, 3$, and report the depth that yields the highest number of successful runs, with the smaller depth reported in the case of a tie. Because searches with different recursion depth can be run in parallel, performance in practice is determined by searches that
are more likely to find a solution; moreover, the required recursion depth to effectively solve a synthesis problem serves as a quantitative measurement of the difficulty of that problem.

An alternative to stochastic search is deterministic enumeration-based search, which can either be brute force (literally enumerate all programs as in [Bansal and Aiken 2006; Massalin 1987]) or guided by a cost function, where a current frontier of candidates are ranked and only the most promising are considered for further exploration ($\lambda^2$ incorporates this strategy). To evaluate the effectiveness of our stochastic search algorithm described in Section 3.1 compared to enumeration-based search, we implement a complete best-first-search bottom-up synthesis procedure for PARA guided by our cost function. We ran PARA under the same settings using this deterministic enumeration procedure instead of the stochastic one on all of our benchmarks, with results labelled by PARA (BFS).

To address question 4, we translate the templates PARA used in successful solutions to the input format of SMYTH, and ask SMYTH to solve the same problem using that translated template. As $\lambda^2$ supports higher-order library procedures, we supply a library of paramorphic combinators to $\lambda^2$ with example-passing axioms and run $\lambda^2$ on the 54 benchmarks with 1) this paramorphism library, and 2) the standard component library and the paramorphism library. Trio’s interface does not provide a way to perform this experiment.

5.2 Results
Detailed experimental results are in Tables 1 and 2 and cactus plots summarizing the running times of all problems solved for each system are in Figure 8. We normalize the CPU time in the cactus plot by multiplying wall-clock time by the number of cores used by each system. PARA solved all but 4 out of the 59 benchmarks reliably, significantly outperforming SMYTH and Trio, which did not solve 26 and 18 benchmarks, respectively. Among the 54 benchmarks that we ported to $\lambda^2$, $\lambda^2$ solves 31 while PARA solves 50. When running PARA on a single core, the system solves all but 1 benchmark that it solves with 8 cores, with typically an order of magnitude longer time. This shows that our embarrassingly parallel algorithm can lead to super-linear speedup on multi-core systems. Among the more difficult problems for which we provided two variants with and without library functions, PARA is able to synthesize complex recursion patterns without the aid of any library functions, while SMYTH, $\lambda^2$ and Trio are only able to solve some of the problems when provided with library functions. For benchmarks that are solved by all systems, SMYTH generally takes less time to find a solution, while Trio solves more problems than any system other than PARA.

When using deterministic best-first-search enumeration, PARA solves 38 of 59 benchmarks. While a more sophisticated deterministic search algorithm could potentially do better than PARA, our enumeration-based search solves more problems than $\lambda^2$, which is also enumeration-based—thus it is likely not easy to find an enumeration-based search comparable to stochastic search. We also note that the stochastic search has the added benefit of requiring only $O(1)$ space for bookkeeping. Figure 9 gives a partial explanation for the strong performance of stochastic search: The number of needed changes to the template to find a solution is almost always less than 15. Previous work on synthesizing straight-line assembly programs has been successful up to 10-15 instructions [Massalin 1987; Schkufza et al. 2013], which requires roughly 50 changes to opcodes, registers and constants starting from an empty program. We likely need harder benchmarks to observe the limits of stochastic search in PARA. Figure 10 gives a whisker plot showing min, max, and distribution by quartile of runtimes of PARA for the problems it solves. The wide distribution and small minimum search times motivate the use of parallelism and restarts to bias PARA towards finding short successful runs.

SMYTH did not solve any of the benchmarks using the automatically generated templates PARA used in successful solutions. We note that Smyth implements the program sketching [Solar-Lezama 2013] synthesis methodology, which is a promising method to express high-level insights from
the programmer and utilize them in synthesis process. However, it is not well-studied whether systematically generated, rather than programmer provided, sets of sketches enhances the problem-solving ability of such systems, and our experiments show no such evidence. \( \lambda^2 \), when provided with the paramorphism library, solves 30 out of the 54 benchmarks. When given both its default library functions and paramorphism combinators, \( \lambda^2 \) does not solve any additional benchmarks besides the ones it already solves using only the default library functions.
We observed a significant negative correlation between the required recursion depth of the problem and the number of problems solved by each system in each category. The results are shown in Table 1 by the required recursion depth for each problem and the percentage of problems solved by each system for each category. The results are shown in Figure 11.

<table>
<thead>
<tr>
<th>Problem</th>
<th>PARA</th>
<th>PARA (BSF)</th>
<th>PARA (Serial)</th>
<th>SYNTH</th>
<th>λ² (std lib)</th>
<th>λ² (paramorphism)</th>
<th>TKK</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool-and</td>
<td>1</td>
<td>0.104</td>
<td>10</td>
<td></td>
<td>0.004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>bool-or</td>
<td>1</td>
<td>0.102</td>
<td>10</td>
<td></td>
<td>0.004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>bool-xor</td>
<td>1</td>
<td>0.101</td>
<td>10</td>
<td></td>
<td>0.004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>bool-impl</td>
<td>1</td>
<td>0.104</td>
<td>10</td>
<td></td>
<td>0.004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>bool-not</td>
<td>2</td>
<td>0.102</td>
<td>10</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-vec</td>
<td>2</td>
<td>0.467</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-speed</td>
<td>1</td>
<td>0.105</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-seven</td>
<td>1</td>
<td>0.124</td>
<td>10</td>
<td></td>
<td>0.004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-sum</td>
<td>1</td>
<td>0.112</td>
<td>10</td>
<td></td>
<td>0.006</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-nil</td>
<td>3</td>
<td>0.115</td>
<td>10</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-exp</td>
<td>3</td>
<td>3.814</td>
<td>6</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-fib</td>
<td>2</td>
<td>4.063</td>
<td>10</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-fatorial</td>
<td>3</td>
<td>3.370</td>
<td>7</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-interpere</td>
<td>2</td>
<td>56.942</td>
<td>7</td>
<td></td>
<td>0.002</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-append</td>
<td>1</td>
<td>0.396</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-concat</td>
<td>2</td>
<td>15.808</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-delete</td>
<td>2</td>
<td>0.114</td>
<td>9</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-even-parity</td>
<td>3</td>
<td>1.183</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-if</td>
<td>1</td>
<td>0.114</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-let</td>
<td>1</td>
<td>0.329</td>
<td>9</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-match</td>
<td>2</td>
<td>0.517</td>
<td>9</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-length</td>
<td>1</td>
<td>0.311</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-nil</td>
<td>2</td>
<td>1.859</td>
<td>2</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-pairwise-swap</td>
<td>3</td>
<td>4.367</td>
<td>3</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-rev</td>
<td>1</td>
<td>17.079</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-share</td>
<td>1</td>
<td>1.840</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-sort</td>
<td>1</td>
<td>1.253</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-sum</td>
<td>2</td>
<td>0.907</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-take</td>
<td>2</td>
<td>14.366</td>
<td>4</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-take-acc</td>
<td>1</td>
<td>0.111</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-collect-leaves</td>
<td>2</td>
<td>7.342</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-count-leaves</td>
<td>3</td>
<td>5.797</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-count-nodes</td>
<td>2</td>
<td>2.823</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-inorder</td>
<td>2</td>
<td>2.516</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-postorder</td>
<td>2</td>
<td>18.360</td>
<td>2</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-preorder</td>
<td>2</td>
<td>4.526</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-node-at-level</td>
<td>3</td>
<td>3.755</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-nil (+)</td>
<td>1</td>
<td>0.963</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-exp (+)</td>
<td>1</td>
<td>1.056</td>
<td>6</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-fib (+)</td>
<td>1</td>
<td>0.975</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>nat-factorial (+)</td>
<td>1</td>
<td>0.202</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-compress (+)</td>
<td>3</td>
<td>37.756</td>
<td>1</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-construct (+)</td>
<td>3</td>
<td>23.881</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>list-sort (insert, -)</td>
<td>3</td>
<td>0.062</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-binarize (+)</td>
<td>2</td>
<td>0.332</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-collect-leaves (+)</td>
<td>2</td>
<td>38.832</td>
<td>8</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-count-leaves (+)</td>
<td>1</td>
<td>10.480</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-count-nodes (+)</td>
<td>1</td>
<td>4.718</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-postorder (+)</td>
<td>3</td>
<td>2.793</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-preorder (+)</td>
<td>2</td>
<td>19.122</td>
<td>2</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>tree-node-at-level (+)</td>
<td>3</td>
<td>6.485</td>
<td>10</td>
<td></td>
<td>0.001</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Table 2. Results with higher-order inputs

<table>
<thead>
<tr>
<th>Problem</th>
<th>PARA (BFS)</th>
<th>PARA (Serial)</th>
<th>SMYTH</th>
<th>Trio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Depth</td>
<td>Time</td>
<td>#Success</td>
<td>Depth</td>
</tr>
<tr>
<td>list-partition</td>
<td>3</td>
<td>25.840</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>list-filter</td>
<td>2</td>
<td>4.645</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>list-fold</td>
<td>1</td>
<td>0.134</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>list-map</td>
<td>1</td>
<td>0.315</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>tree-map</td>
<td>1</td>
<td>26.408</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

For PARA (BFS), PARA (Serial), SMYTH, $\lambda^2$ (paramorphism) and Trio, the percentage of problems solved decreases monotonically when the required recursion depth for PARA increases (Trio very slightly improves at depth 3 over depth 2, but is essentially a tie). The negative correlation is less pronounced for $\lambda^2$ (stdlib), which can be attributed to the fact that the $\lambda^2$ standard library contains many recursive library functions, thus the actual recursion depth $\lambda^2$ needs to successfully synthesize a solution is shallower. In fact, we observed that $\lambda^2$ did not synthesize any nested recursions when used with just its own standard library.

We postulate that the following factors contribute to this observed negative correlation. First, the required recursion depth is a measure of a problem’s difficulty, so we expect any system to perform better on easy problems (low depth) versus hard problems (high depth). Second, during synthesis of a program with higher required recursion depth, more holes are present, which limits the effectiveness of static analysis to propagate information. $\lambda^2$, for example, uses a deductive procedure to discover constraints on the solution, but too many holes can render the analysis ineffective. Third, SMYTH synthesizes top-level general recursion, which in principle expresses a larger set of programs than the properly-nested paramorphisms synthesized by PARA. However, this extra generality incurs a larger search space and potentially less efficient search algorithms, while PARA’s smaller and more structured search space contains solutions to all benchmarks.

As a final experiment, we compare how example-efficient PARA is compared to SMYTH—how few examples are needed to synthesize a solution to a problem? Figure 10, column 1 of [Lubin et al. 2020] shows the minimal number of examples that SMYTH needed to synthesize 34 problems. Using the same input examples, PARA successfully synthesizes 24 of these problems. In fact PARA found solutions for 32 problems that satisfied the input-output examples within the time limit, but 8 were not the intended program. With no bias towards anything other than short solutions and guided purely by observed input/output behavior, it is not surprising PARA requires more examples than other systems to fully constrain the result. We have also shown an upper bound of 16 examples for all the problems in our benchmark suite that PARA successfully solves.
We show some notable synthesis output from PARA to illustrate its ability to synthesize programs with complex recursion patterns. The variables are renamed to improve readability. The following is a solution to the nat-factorial problem found by PARA in 0.172 seconds without library functions:

\[ \text{factorial} = \lambda n : \text{Nat}. \]
\[ \text{para } n \text{ } g_{\text{Zero}} \text{ } g_{\text{Succ}} \text{ where} \]
\[ g_{\text{Zero}} = 1 \]
\[ g_{\text{Succ}} (p_1, s_1) = \text{para } p_1 \text{ } g_{\text{Zero}} \text{ } g_{\text{Succ}} \text{ where} \]
\[ g_{\text{Zero}} = s_1 \]
\[ g_{\text{Succ}} (p_2, s_2) = \text{para } s_2 \text{ } g_{\text{Zero}} \text{ } g_{\text{Succ}} \text{ where} \]
\[ g_{\text{Zero}} = s_1 \]
\[ g_{\text{Succ}} (p_3, s_3) = \text{Succ } s_3 \]

The program consists of three nested loops: the innermost loop computes \( s_1 + s_2 \), the inner two loops compute \( s_1 \times p_1 + s_1 \), and the full loop nest computes \( n! \).

PARA discovered several interesting solutions to the list-rev (list reversal) benchmark. First, PARA finds a solution similar to the list reversal example given in Section 3.2.1 in 2.206 seconds at depth \( D = 1 \), using function types to encode top-down traversal of the input list. At depth \( D = 2 \), PARA finds a solution that uses two nested loops, with the inner loop appending an element to the end of an accumulated list, in 0.747 seconds:

\[ \text{reverse} = \lambda l : \text{ListNat}. \]
\[ \text{para } l \text{ } g_{\text{Nil}} \text{ } g_{\text{Cons}} \text{ where} \]
\[ g_{\text{Nil}} = \text{Nil} \]
\[ g_{\text{Cons}} h_1 (t_1, s_1) = \text{para } s_1 \text{ } g_{\text{Nil}} \text{ } g_{\text{Cons}} \text{ where} \]
\[ g_{\text{Nil}} = \text{Cons } h_1 \text{ Nil} \]
\[ g_{\text{Cons}} h_2 (t_2, s_2) = \text{Cons } h_2 \text{ } s_2 \]

PARA also finds another solution that uses a tuple of two lists as state in 0.711 seconds:

\[ \text{reverse} = \lambda l : \text{ListNat}. \]
\[ \text{para } l \text{ } g_{\text{Nil}} \text{ } g_{\text{Cons}} \text{ where} \]
\[ g_{\text{Nil}} = \text{Tuple Nil l} \]
\[ g_{\text{Cons}} h_1 (t_1, s_1) = \text{para } (\pi_2 s_1) \text{ } g_{\text{Nil}} \text{ } g_{\text{Cons}} \text{ where} \]
\[ g_{\text{Nil}} = \text{Tuple } (\text{Cons } \emptyset t_1) \text{ } t_1 \]
\[ g_{\text{Cons}} h_2 (t_2, s_2) = \text{Tuple } (\text{Cons } h_2 (\pi_1 (\pi_1 s_1))) \text{ } t_2 \]

This program treats the two lists as two stacks, initializing them to an empty list and the input list. The program pops elements from the second stack and pushes them onto the first stack, which eventually becomes the reversal of the input list. Although this program has two nested paras, the inner one is degenerate: the Nil branch is dead code, and the Cons branch extracts the top of the second stack and binds it to \( h_2 \), which is pushed onto the first stack.

6 RELATED WORK

Programming-by-example (PBE) for Domain-Specific Languages. Input-output examples are a simple and accessible way of specifying program behavior. Programming-by-example techniques have been developed to automate domain-specific tasks such as string transformations [Gulwani 2011; Singh and Gulwani 2016], tabular and hierarchical data processing [Chasins et al. 2018; Feng et al. 2017; Wang et al. 2017; Yaghmazadeh et al. 2016], and graphics and visualization [Hempel et al. 2019; Wang et al. 2019]. Some domain-specific PBE approaches support processing specific recursive algebraic data types or their equivalent using a corpus of first-order primitives, such as...
list-processing [Feng et al. 2018]. Those approaches synthesize non-recursive programs that use a predefined set of recursive primitives and are fundamentally more domain-specific than our work.

**PBE for Recursive Functional Programs.** Two early systems, Escher [Albarghouthi et al. 2013] and Myth [Osera and Zdancewic 2015], synthesize recursive functional programs from input-output examples. Escher [Albarghouthi et al. 2013] is a bottom-up synthesis procedure for an untyped first-order language with general recursion. Available data types are restricted to a predefined set of base types. Myth [Osera and Zdancewic 2015] is a top-down deductive synthesis procedure for a typed higher-order language with user-defined algebraic data types. It performs proof search to produce terms that satisfy a given set of input-output examples. Both Escher and Myth require input-output examples to be *trace-complete*, i.e. the input-output examples must include the input/output pairs of all recursive calls. Writing trace-complete examples is cumbersome and requires some intuition for the internal computations of the program to be synthesized.

Burst [Miltner et al. 2022] uses bottom-up synthesis that handles general recursion without requiring trace-completeness by using *angelic execution*, executing unknown recursive calls under angelic semantics during synthesis and then refining the specification if the synthesized program is incorrect under normal semantics. The procedure repeats until a correct program is found.

All of the above-mentioned systems synthesize a single recursive function with recursive calls to the top-level function itself. In practice, however, solutions to many problems are more naturally expressed with mutually recursive functions. Even though a single general recursion at top-level is in theory capable of encoding arbitrary mutually recursive functions, currently such complex general recursive functions are not practically synthesized by these systems, which poses difficulties in applying these approaches to more challenging problems.

**PBE for data structure transformations.** $\lambda^2$ [Feser et al. 2015] synthesizes functional programs that transform data structures using higher-order combinators such as fold, map and filter. Similar to templates used by Para, $\lambda^2$ generates multi-hole hypotheses using these combinators. It then infers the input-output examples for the holes via deductive reasoning. $\lambda^2$ is notable for its ability to synthesize nested recursive transformations. In practice, we find that $\lambda^2$’s approach is limited to deducing examples one-hole-at-a-time. When there are multiple holes in a hypothesis but no example adequately constrains any single hole, $\lambda^2$ falls back to exhaustive enumeration. There are also specific requirements on the provided input-output examples for the deduction procedure to be effective. For example, while $\lambda^3$ is generally able to propagate input-output examples of a map hypothesis into its functional argument, the same is possible for a fold hypothesis only if there are pairs of examples that differ by only appending or prepending one element to the input sequence.

In our experiments using randomly-generated examples, we find that $\lambda^2$ often only propagates only the base case example for fold-like combinators including fold and our paramorphism combinators. Para on the other hand can take advantage of simultaneous constraints on multiple holes by probabilistically making multiple changes, even if each of the changes individually does not decrease the cost. Para also uses fold-like combinators effectively with fewer restrictions on the input-output examples.

**Program Sketching.** Sketching [Solar-Lezama 2013] is a synthesis methodology that synthesizes concrete implementations by completing program sketches, i.e. programmer-provided partial programs with holes. Sketching is a promising method to express high-level insights from the programmer and utilize them in synthesis process. We note that it is possible to provide such systems with a systematically generated set of sketches that describes different recursion shapes, in hope of improving their ability to solve more difficult problems. Synapse has applied this approach to a number of non-recursive synthesis problems[Bornholt et al. 2016]. However, to our knowledge such an approach has not been well-studied for recursive programs, which are generally more difficult to synthesize.
Sketch [Solar-Lezama 2013] pioneered the sketching approach, synthesizing programs in an imperative, C-like language by translating input-output examples of the synthesis problem to a logical formula that constrains admissible hole values and relying on an external SAT or SMT solver to find hole value assignments. Rosette develops a similar approach for the untyped functional language Racket [Tolak and Bodik 2013]. This approach inherently limits the types of holes to those that external solvers handle well. In practice, only booleans and integers are supported. Syntrec [Inala et al. 2017] suggests that it is possible to encode user-defined algebraic data types in such a framework. However, such an encoding is limited to some subset of all possible well-typed terms for any given hole, such as bounding the number of constructors by a pre-determined limit.

SMYTH [Lubin et al. 2020] is a descendant of Myth with support for program sketching, allowing users to provide a custom template with multiple holes. SMYTH also removes the trace-completeness requirement from Myth via live bidirectional evaluation, which propagates examples backward through partially evaluated incomplete programs. A key difference between SMYTH and Para is that SMYTH is design to support general recursion while Para specifically targets primitive recursion. As discussed in Section 5, the extra generality of SMYTH is not exploited by any of the benchmarks in the literature while also likely making it more challenging for SMYTH to solve the harder problems.

Leon [Kneuss et al. 2013] and Synquid [Polikarpova et al. 2016] complete program sketches from logical specifications. Leon synthesizes Scala functions given pre-conditions and post-conditions by combining a term generation system, a verifier, and a conditional abduction algorithm to generate recursive program fragments. Synquid accepts logical specifications in the form of refinement types of the target program. Synquid employs a Myth-like proof search procedure, relying on an external SMT solver to perform refinement type checking. These logical specification-based synthesizers can be applied to PBE tasks by encoding input-output examples as a conjunction of propositions. However, the underlying techniques are not necessarily tailored to specifications of such structure. For example, Synquid requires the input specification to be inductive, which is similar to the trace-completeness requirement. See [Lubin et al. 2020] for experiments comparing the performance of Leon and Synquid applied to PBE tasks versus dedicated PBE synthesizers.

SyRup uses version space algebra to hypothesize pairs of recursive programs and execution traces, using the consistency of a program with a trace to guide progress [Yuan et al. 2023]. In testing SyRup, we found that it solves the bool and nat benchmarks but only one list and one tree benchmark. SyRup relies on having some trace complete examples to do well; with only randomly generated examples (our scenario), [Yuan et al. 2023] showed that SyRup does not consistently outperform SMYTH. Overall, the different goals of SyRup—minimizing the number of examples while requiring some trace completeness—makes a fair comparison with Para difficult.

Stochastic Search. StokE [Schkuza et al. 2013] applies stochastic MCMC search to the superoptimization task for 64-bit x86 instruction sequences. StokE synthesizes optimized versions of a given target program under some performance metric. Effective application of the approach is limited to loop-free programs. While our cost function is different, we also measure the differences between the output of a candidate program and the desired output.

Genetic Programming (GP) [Koza 1994] stochastically evolves a program to improve its fitness for a particular task. GP has been applied to evolve general recursive functional expressions that satisfy given input-output examples [Moraglio et al. 2012; Nishiguchi and Fujimoto 1998; Wong and Leung 1996]. However, evolving non-trivial general recursive functions using GP has proven difficult in practice [Agapitos and Lucas 2006; Agapitos et al. 2017; Alexander and Zacher 2014].

In [Yu 2001; Yu and Clack 1998], typed programs are evolved in primitive-recursive forms for lists using λ-abstractions and a fold operator. In [Binard and Felty 2008] programs in System-F [Girard 1971, 1972; Reynolds 1974], a total (always terminating) functional language with polymorphic types, are evolved using similar search techniques. System-F includes programs with much higher time
complexity than primitive recursive functions. Experimental evaluation of the above approaches is limited and it is unclear how well these approaches apply to a broad class of programming problems.

In [Swan et al. 2019], GP-based stochastic synthesis of implicitly recursive programs utilizing recursion schemes is explored by fixing a catamorphism [Meijer et al. 1991] template at top-level. Ant programming or random search is used to synthesize expressions for each constructor case in the template. Because only a single catamorphism combinator at top-level is used, we expect this approach to have similar issues as aforementioned systems with top-level only general recursion when applied to difficult problems.

Synthesis of Auxiliary Functions. [Eguchi et al. 2018] extends SYNQUID to synthesize functional programs with recursive auxiliary functions. Their approach employs top-level templates with auxiliary function holes. The refinement types of the holes are inferred given the top-level refinement type of the target program, and then SYNQUID is used to find instantiations of the auxiliary functions. The templates use a restricted syntax, which is less flexible in terms of where holes are allowed to appear compared to our template syntax. For example, the argument to a pattern matching construct is not allowed to contain holes. They employ two predefined top-level templates, a fold-like (catamorphism) template and a divide-conquer-type template, rather than systematically generated templates. These templates are not nested so at most two levels of recursion are possible (one from the top-level template and one from auxiliary functions synthesized by SYNQUID).

Cypress [Itzhaky et al. 2021] synthesizes imperative heap-manipulating programs with auxiliary recursive procedures. Cypress extends a deductive synthesis framework with cyclic proofs and uses abduction during proof search to discover potential application of cyclic reasoning. Recursive auxiliary procedures naturally arise from cyclic derivations, with backlinks in the derivations corresponding to recursive calls. Because of the very different target programs and synthesis techniques, the relationship, if any, between Cypress and our work is unclear.

7 CONCLUSIONS AND LIMITATIONS

We have presented a new program synthesis technique for recursive functions over algebraic data types based on paramorphisms. By splitting the synthesis problem into selecting a skeleton of nested paramorphisms with holes and synthesizing non-recursive terms to fill the holes, we are able to reuse simple and effective stochastic search techniques to synthesize complex recursive programs. We have shown by experiment that an implementation of our approach is able to synthesize all the problems handled by the current state of the art as well as substantially harder problems.

Our method is not without limitations. Primitive recursion, while well-matched to algebraic data types, is not as expressive as general recursion. Some programming patterns, such as worklist algorithms on graphs, would be awkward or even impossible to express with paramorphisms. We have shown that the benchmark examples used for recursive program synthesis in the literature are primitive recursive. Thus, it appears that program synthesis methods for problems that truly require general recursion have yet to be developed, and it is an open question whether practical methods for synthesizing general recursive programs exist for problems that actually require general recursion.

REFERENCES


Received 2023-11-16; accepted 2024-03-31