Synthesis of Recursive ADT Transformations from Reusable Templates

Jeevana Priya Inala[1(✉)], Nadia Polikarpova[1], Xiaokang Qiu[2], Benjamin S. Lerner[3], and Armando Solar-Lezama[1]

[1] MIT, Cambridge, USA
{jinala,polikarn,asolar}@csail.mit.edu
[2] Purdue University, West Lafayette, USA
xkqiu@purdue.edu
[3] Northeastern University, Boston, USA
blerner@ccs.neu.edu

Abstract. Recent work has proposed a promising approach to improving scalability of program synthesis by allowing the user to supply a syntactic template that constrains the space of potential programs. Unfortunately, creating templates often requires nontrivial effort from the user, which impedes the usability of the synthesizer. We present a solution to this problem in the context of recursive transformations on algebraic data-types. Our approach relies on *polymorphic synthesis constructs*: a small but powerful extension to the language of syntactic templates, which makes it possible to define a program space in a concise and highly reusable manner, while at the same time retains the scalability benefits of conventional templates. This approach enables end-users to reuse predefined templates from a library for a wide variety of problems with little effort. The paper also describes a novel optimization that further improves the performance and the scalability of the system. We evaluated the approach on a set of benchmarks that most notably includes desugaring functions for lambda calculus, which force the synthesizer to discover Church encodings for pairs and boolean operations.

1 Introduction

Recent years have seen remarkable advances in tools and techniques for automated synthesis of recursive programs [1,4,8,13,16]. These tools take as input some form of *correctness specification* that describes the intended program behavior, and a set of building blocks (or *components*). The synthesizer then performs a search in the space of all programs that can be built from the given components until it finds one that satisfies the specification. The biggest obstacle to practical program synthesis is that this search space grows extremely fast with the size of the program and the number of available components. As a result, these tools have been able to tackle only relatively simple tasks, such as textbook data structure manipulations.

Syntax-guided synthesis (SyGuS) [2] has emerged as a promising way to address this problem. SyGuS tools, such as SKETCH [18] and Rosette [20,21]

A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 247–263, 2017.
DOI: 10.1007/978-3-662-54577-5_14

leverage a user-provided syntactic *template* to restrict the space of programs the synthesizer has to consider, which improves scalability and allows SyGus tools to tackle much harder problems. However, the requirement to provide a template for every synthesis task significantly impacts usability.

This paper shows that, at least in the context of recursive transformations on algebraic data-types (ADTs), it is possible to get the best of both worlds. Our first contribution is a new approach to making syntactic templates highly reusable by relying on *polymorphic synthesis constructs* (*PSC*s). With *PSC*s, a user does not have to write a custom template for every synthesis problem, but can instead rely on a generic template from a library. Even when the user does write a custom template, the new constructs make this task simpler and less error-prone. We show in Sect. 4 that all our 23 diverse benchmarks are synthesized using just 4 different generic templates from the library. Moreover, thanks to a carefully designed type-directed expansion mechanism, our generic templates provide the same performance benefits during synthesis as conventional, program-specific templates. Our second contribution is a new optimization called *inductive decomposition*, which achieves asymptotic improvements in synthesis times for large and non-trivial ADT transformations. This optimization, together with the user guidance in the form of reusable templates, allows our system to attack problems that are out of scope for existing synthesizers.

We implemented these ideas in a tool called SynthRec, which is built on top of the open source Sketch synthesis platform [19]. Our tool supports expressive correctness specifications that can use arbitrary functions to constrain the behavior of ADT transformations. Like other expressive synthesizers, such as Sketch [18] and Rosette [20,21], our system relies on exhaustive bounded checking to establish whether a program candidate matches the specification. While this does not provide correctness guarantees beyond a bounded set of inputs, it works well in practice and allows us to tackle complex problems, for which full correctness is undecidable and is beyond the state of the art in automatic verification. For example, our benchmarks include desugaring functions from an abstract syntax tree (AST) into a simpler AST, where correctness is defined in terms of interpreters for the two ASTs. As a result, our synthesizer is able to discover Church encodings for pairs and booleans, given nothing but an interpreter for the lambda calculus. In another benchmark, we show that the system is powerful enough to synthesize a type constraint generator for a simple programming language given the semantics of type constraints. Additionally, several of our benchmarks come from transformation passes implemented in our own compiler and synthesizer.

2 Overview

In this section, we use the problem of desugaring a simple language to illustrate the main features of SynthRec. Specifically, the goal is to synthesize a function `dstAST desugar(srcAST src) {...}`, which translates an expression in source AST into a semantically equivalent expression in destination AST. Data type definitions for the two ASTs are shown in Fig. 1: the type `srcAST` has five *variants* (two

```
adt srcAST{
  NumS{ int v; }
  TrueS{ }
  FalseS{ }
  BinaryS{ opcode op; srcAST a; srcAST b;}
  BetweenS{ srcAST a; srcAST b; srcAST c;}}

adt dstAST{
  NumD{ int v; }
  BoolD{ bit v; }
  BinaryD{ opcode op; dstAST a; dstAST b;}}

adt opcode{ AndOp{} OrOp{} LtOp{}}
```

Fig. 1. ADTs for two small expression languages

of which are recursive), while dstAST has only three. In particular, the source language construct BetweenS(a, b, c), which denotes a < b < c, has to be desugared into a conjunction of two inequalities. Like case classes in Scala, data type variants in SyntRec have named fields.

Specification. The first piece of user input required by the synthesizer is the specification of the program's intended behavior. In the case of desugar, we would like to specify that the desugared AST is semantically equivalent to the original AST, which can be expressed in SyntRec using the following constraint:

```
assert( srcInterpret (exp) == dstInterpret(desugar(exp)) )
```

This constraint states that interpreting an arbitrary source-language expression exp (bounded to some depth) must be equivalent to desugaring exp and interpreting the resulting expression in the destination language. Here, srcInterpret and dstInterpret are regular functions written in SyntRec and defined recursively over the structure of the respective ASTs in a straightforward manner. As we explain in Sect. 3.4, our synthesizer contains a novel optimization called *inductive decomposition* that can take advantage of the structure of the above specification to significantly improve the scalability of the synthesis process.

Templates. The second piece of user input required by our system is a syntactic *template*, which describes the space of possible implementations. The template is intended to specify the high-level structure of the program, leaving low-level details for the system to figure out. In that respect, SyntRec follows the SyGuS paradigm [2]; however, template languages used in existing SyGuS tools, such as Sketch or Rosette, work poorly in the context of recursive ADT transformations.

For example, Fig. 2 shows a template for desugar written in Sketch, the predecessor of SyntRec. It is useful to understand this template as we will show, later, how the new language features in SyntRec allow us to write the same template in a concise and reusable manner. This template uses three kinds of *synthesis constructs* already existing in Sketch: a *choice* (**choose**$(e_1, ..., e_n)$) must be replaced with one of the expressions $e_1, \ldots, e_n$; a *hole* (**??**) must be replaced with an integer or a boolean constant; finally, a *generator* (such as rcons) can be thought of as a macro, which is inlined on use, allowing the synthesizer to make different choices for every invocation[1]. The task of the synthesizer is to

[1] Recursive generators, such as rcons, are unrolled up to a fixed depth, which is a parameter to our system.

```
dstAST desugar(srcAST src){
  switch(src) {
  case NumS:
    return rcons(src.v);
  ... /* Some cases are elided */
  case BinaryS:
    dstAST a = desugar(src.a), b = desugar(src.b);
    return rcons(choose(a, b, src.op));
  case BetweenS:
    dstAST a = desugar(src.a), b = desugar(src.b),
           c = desugar(src.c);
    return rcons(choose(a, b, c));
}}
```

```
generator dstAST rcons(fun e) {
  if (??) return e();
  if (??) {
    int val = choose(e(), ??);
    return new NumD(v = val); }
  if (??) {
    bit val = choose(e(), ??);
    return new BoolD(v = val);}
  if (??) {
    dstAST a = rcons(e);
    dstAST b = rcons(e);
    opcode op = choose(e(), new AndOp(),...,
                       new LtOp());
    return new BinaryD(op = op, a= a, b = b);}
}
```

Fig. 2. Template for desugar in SKETCH

fill in every choice and hole in such a way that the resulting program satisfies the specification.

The template in Fig. 2 expresses the intuition that desugar should recursively traverse its input, src, replacing each node with some subtree from the destination language. These destination subtrees are created by calling the recursive, higher-order generator rcons (for "recursive constructor"). rcons(e) constructs a nondeterministically chosen variant of dstAST, whose fields, depending on their type, are obtained either by recursively invoking rcons, by invoking e (which is itself a generator), or by picking an integer or boolean constant. For example, one possible instantiation of the template rcons(**choose**(x, y, src.op))[2] can lead to **new** BinaryD(op = src.op, a = x, b = **new** NumD(5)). Note that the template for desugar provides no insight on how to actually encode each node of scrAST in terms of dstAST, which is left for the synthesizer to figure out. Despite containing so little information, the template is very verbose: in fact, more verbose than the full implementation! More importantly, this template cannot be reused for other synthesis problems, since it is specific to the variants and fields of the two data types. Expressing such a template in Rosette will be similarly verbose.

Reusable Templates. SYNTREC addresses this problem by extending the template language with *polymorphic synthesis constructs* (*PSC*s), which essentially support parametrizing templates by the structure of data types they manipulate. As a result, in SYNTREC the end user can express the template for desugar with a single line of code:

```
dstAST desugar(srcAST src) { return recursiveReplacer(src, desugar);}
```

Here, recursiveReplacer is a reusable generator defined in a library; its code is shown in Fig. 3. When the user invokes recursiveReplacer(src,desugar), the body

[2] When an expression is passed as an argument to a higher-order function that expects a function parameter such as rcons, it is automatically casted to a *generator lambda* function. Hence, the expression will only be evaluated when the higher-order function calls the function parameter and each call can result in a different evaluation.

```
1  generator T recursiveReplacer<T, Q>(Q src,
2                fun rec) {
3    switch(src){
4      case?:
5          T[ ] a = map(src.fields?, rec);
6          return rcons(choose(a[??],
7                       field(src)));
8  }}}
9  generator T rcons<T>(fun e) {
10   if (??) return e();
11   else return new cons?(rcons(e));
12 }
13 generator T field<T,S>(S e) {
14   return (e.fields?) [??];
15 }
```

```
1  dstAST desugar(srcAST src) {
2    switch(src) {
3      case NumS: return new NumD(v = src.v);
4      case TrueS: return new BoolD(v = 1);
5      case FalseS: return new BoolD(v = 0);
6      case BinaryS:
7        dstAST[2] a = {desugar(src.a), desugar(src.b)};
8        return new BinaryD(op = src.op, a = a[1],
9                           b = a[2]);
10     case BetweenS:
11       dstAST[3] a = {desugar(src.a), desugar(src.b),
12                      desugar(src.c)};
13       return new BinaryD(op = new AndOp(),
14         a = new BinaryD(op = new LtOp(), a = a[0],
15                         b = a[1])
16         b = new BinaryD(op = new LtOp(), a = a[1],
17                         b = a[2]));
18 }}
```

Fig. 3. Left: generic template for recursiveReplacer. Right: solution to the running example

of the generator is specialized to the surrounding context, resulting in a template very similar to the one in Fig. 2. Unlike the template in Fig. 2, however, recursiveReplacer is not specific to srcAST and dstAST, and can be reused with no modifications to synthesize desugaring functions for other languages, and even more general recursive ADT transformations. Crucially, even though the reusable template is much more concise than the SKETCH template, it does not increase the size of the search space that the synthesizer has to consider, since all the additional choices are resolved during type inference. Figure 3 also shows a compacted version of the solution for desugar, which SYNTREC synthesizes in about 8s. The rest of the section gives an overview of the *PSC*s used in Fig. 3.

Polymorphic Synthesis Constructs. Just like a regular synthesis construct, a *PSC* represents a set of potential programs, but the exact set depends on the context and is determined by the types of the arguments to a *PSC* and its expected return type. SYNTREC introduces four kinds of *PSC*s.

1. A ***Polymorphic Generator*** is a polymorphic version of a SKETCH generator. For example, recursiveReplacer is a polymorphic generator, parametrized by types T and Q. When the user invokes recursiveReplacer(src,desugar), T and Q are instantiated with dstAST and srcAST, respectively.
2. ***Flexible Pattern Matching*** (**switch**(x) **case?**: e) expands into pattern matching code specialized for the type of x. In our example, once Q in recursiveReplacer is instantiated with srcAST, the **case?** construct in Line 4 expands into five cases (**case** NumS, ..., **case** BetweenS) with the body of **case?** duplicated inside each of these cases.
3. ***Field List*** (e.**fields?**) expands into an array of all fields of type τ in a particular variant of e, where τ is derived from the context. Going back to Fig. 3, Line 5 inside recursiveReplacer maps a function rec over a field list src.**fields?**; in our example, rec is instantiated with desugar, which takes an input of type srcAST.

Hence, SYNTREC determines that src.**fields?** in this case denotes all fields of type srcAST. Note that this construct is expanded differently in each of the five cases that resulted from the expansion of **case?**. For example, inside **case** NumS, this construct expands into an empty array (NumS has no fields of type srcAST), while inside **case** BetweenS, it expands into the array {src.a, src.b, src.c}.

4. ***Unknown Constructor*** (**new cons?**(e_1, ..., e_n)) expands into a constructor for some variant of type τ, where τ is derived from the context, and uses the expressions $e_1, \ldots, e_n$ as the fields. In our example, the auxiliary generator rcons uses an unknown constructor in Line 11. When rcons is invoked in a context that expects an expression of type dstAST, this unknown constructor expands into **choose**(**new** NumD(...), **new** BoolD(...), **new** BinaryD(...)). If instead rcons is expected to return an expression of type opcode, then the unknown constructor expands into **choose**(**new** AndOp(),...,**new** LtOp()). If the expected type is an integer or a boolean, this construct expands into a regular SKETCH hole (**??**).

Even though the language provides only four *PSC*s, they can be combined in novel ways to create richer polymorphic constructs that can be used as library components. The generators field and rcons in Fig. 3 are two such components.

The field component expands into an arbitrary field of type τ, where τ is derived from the context. Its implementation uses the *field list PSC* to obtain the array of all fields of type τ, and then accesses a random element in this array using an integer hole. For example, if field (e) is used in a context where the type of e is BetweenS and the expected type is srcAST, then field (e) expands into {e.a, e.b, e.c}[**??**] which is semantically equivalent to **choose**(e.a, e.b, e.c).

$$
\begin{aligned}
P &:= \{adt_i\}_i \ \ \{f_i\}_i \\
adt &:= \textbf{adt}\ name\{\ \ variant_1 \ldots variant_n\ \} \\
variant &:= name\,\{l_1 : \tau_1 \ldots\ l_n : \tau_n\} \\
\theta &:= \tau \mid T \mid \theta[\,] \mid fun \mid \theta_1 \to \theta_2 \\
\tau &:= prim \mid name \mid \{l_i : \tau_i\}_{i<n} \\
&\quad \mid \textstyle\sum name_i\,\{l^i_k : \tau^i_k\}_{k<n_i} \\
prim &:= \textbf{bit} \mid \textbf{int} \\
f &:= \ \overline{f} \mid \hat{f} \mid \hat{\hat{f}} \\
\overline{f} &:= \tau_{out}\ name\,(\{x_i : \tau_i\}_i)\ \ e \\
\hat{f} &:= \textbf{generator}\ \tau_{out}\ name\,(\{x_i : \tau_i\}_i)\ e \\
\hat{\hat{f}} &:= \textbf{generator}\ \ \theta_{out}\ name\langle\{T_i\}_i\rangle\,(\{x_i : \theta_i\}_i)\ \ e \\
e &:= \ \overline{e} \mid \hat{e} \mid \hat{\hat{e}} \\
\overline{e} &:= \ \ x \mid \textbf{let}\ x : \theta = e_1\ \textbf{in}\ \ e_2 \mid f(e) \\
&\quad \mid \textbf{switch}\,(x)\,\{\ \textbf{case}\ \ name_i : e_i\ \}_i \\
&\quad \mid \ e.l \mid \textbf{new}\ name(\{l_i = e_i\}_i) \\
&\quad \mid \ \{\{e_i\}_i\} \mid e_1[e_2] \mid \textbf{assert}(e) \\
\hat{e} &:= \ \ ?? \mid \textbf{choose}(\{e_i\}_i) \mid \ \hat{f}(e) \\
\hat{\hat{e}} &:= \ \ \hat{\hat{f}}(e) \mid \textbf{new cons?}(\{e_i\}_i) \\
&\quad \mid \ e.\textbf{fields?} \mid \textbf{switch}(x)\{\textbf{case?} : e\}
\end{aligned}
$$

Fig. 4. Kernel language

The rcons component is a polymorphic version of the recursive constructor for dstAST in Fig. 2, and can produce ADT trees of any type up to a certain depth. Note that since rcons is a polymorphic generator, each call to rcons in the argument to the unknown constructor (Line 11) is specialized based on the type required by that constructor and can make different nondeterministic choices. Similarly, it is possible to create other generic constructs such as iterators over arbitrary data structures. Components such as these are expected to be provided by expert users, while end users treat them in the same way as the built-in *PSC*s. The next section gives a formal account of the SYNTREC's language and the synthesis approach.

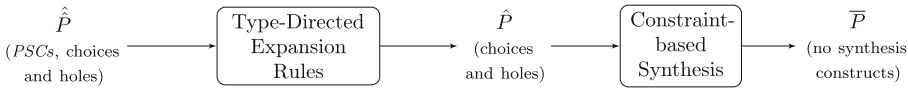
3 SYNTREC Formally

3.1 Language

Figure 4 shows a simple kernel language that captures the relevant features of SYNTREC. In this language, a program consists of a set of ADT declarations followed by a set of function declarations. The language distinguishes between a standard function $\overline{f}$, a generator $\hat{f}$ and a *polymorphic generator* $\hat{\hat{f}}$. Functions can be passed as parameters to other functions, but they are not entirely first-class citizens because they cannot be assigned to variables or returned from functions. Function parameters lack type annotations and are declared as type fun, but their types can be deduced from inference. Similarly, expressions are divided into standard expressions that does not contain any unknown choices ($\overline{e}$), existing synthesis constructs in SKETCH ($\hat{e}$), and the new *PSC*s ($\hat{\hat{e}}$). The language also has support for arrays with expressions for array creation ($\{e_1, e_2, ..., e_n\}$) and array access ($e_1[e_2]$). An array type is represented as $\theta[\,]$. In this formalism, we use the Greek letter τ to refer to a fully concrete type and θ to refer to a type that may involve type variables. The distinction between the two is important because *PSC*s can only be expanded when the types of their context are known. We formalize ADTs as tagged unions $\tau = \sum variant_i$, where each of the variants is a record type $variant_i = name_i \left\{ l_k^i : \tau_k^i \right\}_{k<n_i}$. Note that ADTs in SYNTREC are not polymorphic. The notation $\{a_i\}_i$ is used to denote the $\{a_1, a_2, ...\}$.

3.2 Synthesis Approach

Given a user-written program $\hat{\hat{P}}$ that can potentially contain *PSC*s, choices and holes, and a specification, the synthesis problem is to find a program $\overline{P}$ in the language that only contains standard expressions ($\overline{e}$) and functions ($\overline{f}$). SYNTREC solves this problem using a two step approach as shown below:



First, SYNTREC uses a set of expansion rules that uses bi-directional type checking to eliminate the *PSC*s. The result is a program that only contains choices and holes. The second step is to use a constraint-based approach to solve for these choices. The next subsections will present each of these steps in more detail.

3.3 Type-Directed Expansion Rules

We will now formalize the process of specializing and expanding the *PSC*s into sets of possible expressions. We should first note that the expansion and the specialization of the different *PSC*s interact in complex ways. For example, for the **case?** construct in the running example, the system cannot determine which cases to generate until it knows the type of src, which is only fixed once the

polymorphic generator for recursiveReplacer is specialized to the calling context. On the other hand, if a *polymorphic generator* is invoked inside the body of a **case?** (like rcons in the running example), we may not know the types of the arguments until after the **case?** is expanded into separate cases. Because of this, type inference and expansion of the *PSC*s must happen in tandem.

We formalize the process of expanding *PSC*s using two different kinds of judgements. The *typing judgement* $\Gamma \vdash e : \theta$ determines the type of an expression by propagating information bottom-up from sub-expressions to larger expressions. On the other hand, *PSC*s cannot be type-checked in a bottom-up manner; instead, their types must be inferred from the context. The *expansion judgment* $\Gamma \vdash e \xrightarrow{\theta} e'$ expands an expression e involving *PSC*s into an expression e' that does not contain *PSC*s (but can contain choices and holes). In this judgment, θ is used to propagate information top-down and represents the type required in a given context; in other words, after this expansion, the typing judgement $\Gamma \vdash e' : \theta$ must hold. We are not the first to note that bi-directional typing [15] can be very useful in pruning the search space for synthesis [13,16], but we are the first to apply this in the context of constraint-based synthesis and in a language with user-provided definitions of program spaces.

$$FUN \quad \frac{\Gamma; \{x_i : \tau_i\}_{i<n} \vdash e \xrightarrow{\tau_o} e'}{\Gamma \vdash \tau_o\ f\ (\{x_i : \tau_i\}_{i<n})\ e \xrightarrow{\perp} \tau_o\ f\ (\{x_i : \tau_i\}_{i<n})\ e'}$$

$$FL \quad \frac{\Gamma \vdash e : \{l_i : \tau_i\}_{i<n} \quad \Gamma \vdash e \xrightarrow{\{l_i:\tau_i\}_{i<n}} e' \quad \{\tau_{i_j} = \tau_0\}_j \quad (\tau_0[\,] = \tau)}{\Gamma \vdash e.\,\textbf{fields?} \xrightarrow{\tau} \{\{e'.l_{ij}\}_j\}}$$

$$FPM \quad \frac{\Gamma = \left(\Gamma'; x\ :\ \sum\ name_i\ \{l_k^i : \tau_k^i\}_{k<n_i}\right) \qquad \left\{\left(\Gamma'; x\ :\ \{l_k^i : \tau_k^i\}_{k<n_i}\right) \vdash e \xrightarrow{\theta} e_i\right\}_i}{\Gamma \vdash \textbf{switch}\,(x)\,\{\ \textbf{case?} : e\ \} \xrightarrow{\theta} \textbf{switch}\,(x)\,\{\ \textbf{case}\ name_i : e_i\}_i}$$

$$UC1 \quad \frac{\tau = \Sigma name_i\ \{l_k^i : \tau_k^i\}_{k<n_i} \qquad e_1 \xrightarrow{\tau_k^i} e_{1_k^i} \ldots e_m \xrightarrow{\tau_k^i} e_{m_k^i}}{\Gamma \vdash\ \textbf{new cons?}\,(e_1 \ldots e_m) \xrightarrow{\tau}\ \textbf{choose}\left(\left\{\textbf{new}\ name_i\left(\left\{l_k^i = \textbf{choose}\left(\{e_{r_k}^i\}_{r<m}\right)\right\}_{k<n_i}\right)\right\}_i\right)}$$

$$UC2 \quad \frac{\tau = prim}{\Gamma \vdash\ \textbf{new cons?}\,(e_1 \ldots e_m) \xrightarrow{\tau}\ ??}$$

$$PG \quad \frac{\begin{array}{c}\theta_{out}\ \ \hat{f}\,\langle\{T_i\}\rangle\,(\{p_i : \theta_i\}_i)\ \ e \qquad \Gamma \vdash e_i : \tau_i^{in} \quad for\ i < k \\ S = \textsf{Unify}\left(\{(\theta_{out},\ \theta)\} \cup \{(\theta_i, \tau_i^{in})\}_{i<k}\right) \\ e_i \xrightarrow{S(\theta_i)} e_i' \quad for\ i \le k+n \qquad e[\{e_i'/p_i\}_i] \xrightarrow{S(\theta)} e'\end{array}}{\hat{f}\,(e_0 \ldots e_k \ldots e_{k+n}) \xrightarrow{\theta} e'}$$

Fig. 5. Expansion rules for various language constructs

The expansion rules for functions and *PSC*s are shown in Fig. 5. At the top level, given a program P, every function in P is transformed using the expansion rule FUN. The body of the function is expanded under the known output type of the function. The most interesting cases in the definition of the expansion judgment correspond to the *PSC*s as outlined below. The expansion rules for the other expressions are straightforward and are elided for brevity.

Field List. The rule FL shows how a *field list* is expanded. If the required type is an array of τ_0, then this *PSC* can be expanded into an array of all fields of type τ_0.

Flexible Pattern Matching. For each case, the body of **case?** is expanded while setting x to a different type corresponding to each variant $name_i \left\{l_k^i : \tau_k^i\right\}_{k<n_i}$ as shown in the rule FPM. Here, the argument to **switch** is required to be a variable so that it can be used with a different type inside each of the different cases. Note that each case is expanded independently, so the synthesizer can make different choices for each e_i.

Unknown Constructor. If the required type is an ADT, the rule UC1 expands the expressions passed to the *unknown constructor* based on the type of each field of each variant of the ADT and uses the resulting expressions to initialize the fields in the relevant constructor. It returns a **choose** expression with all these constructors as the arguments. If the required type is a primitive type (int or bit), the unknown constructor is expanded into a SKETCH hole by the rule UC2.

Polymorphic Generator Calls. When the expansion encounters a call to a *polymorphic generator*, the generator will be expanded and specialized according to the PG rule. When a generator is called with arguments $\{e_i\}_i$, we can separate the arguments into expressions that can be typed using the standard typing judgement, and expressions such as **new cons?** (...) that cannot. In the rule, we assume, without loss of generality, that the first k expressions can be typed and the reminder cannot. The basic idea behind the expansion is as follows. First, the rule obtains the types of the first k arguments and unifies them with the types of the formal parameters of the function to get a type substitution S. The arguments to the original call are expanded with our improved knowledge of the types, and the body of the generator is then inlined and expanded in turn. The actual implementation also keeps track of how many times each generator has been inlined and replaces the generator invocation with **assert** false when the inlining bound has been reached.

The above expansion rules fail if a type variable is encountered in places where a concrete type is expected, and in such cases the system will throw an error. For example, expressions such as field (field (e)), where field is as defined in Fig. 3, cannot by type-checked in our system because the expected type of the inner field call cannot be determined using top-down type propagation.

3.4 Constraint-Based Synthesis

Once we have a program with a fixed number of integer unknowns, the synthesis problem can be encoded as a constraint $\exists\phi.\ \forall\sigma.\ P(\phi, \sigma)$ where ϕ is a *control vector* describing the set of choices that the synthesizer has to make, σ is the input state of the program, and $P(\phi, \sigma)$ is a predicate that is true if the program satisfies its specification under input σ and control ϕ. Our system follows the standard approach of unrolling loops and inlining recursive calls to derive P and uses counterexample guided inductive synthesis (CEGIS) to solve this doubly

quantified problem [18]. For readers unfamiliar with this approach, the most relevant aspect from the point of view of this paper is that the doubly quantified problem is reduced to a sequence of inductive synthesis steps. At each step, the system generates a solution that works for a small set of inputs, and then checks if this solution is in fact correct for all inputs; otherwise, it generates a counter-example for the next inductive synthesis step.

Applying the standard approach can, however, be problematic in our context especially with regards to inlining recursive calls. For instance, consider the example from Sect. 2. Here, the function desugar that has to be synthesized is a recursive function. If we were to inline all the recursive calls to desugar, then a given concrete value for the input σ such as BetweenS(a = NumS(...), b = BinaryS(...), ...) , will exercise multiple cases within desugar (BetweenS, NumS and BinaryS for the example). This is problematic in the context of CEGIS, because at each inductive synthesis step the synthesizer has to jointly solve for all these variants of desugar which greatly hinders scalability when the source language has many variants.

3.5 Inductive Decomposition

The goal of this section is to leverage the inductive specification to potentially avoid inlining the recursive calls to the synthesized function. This idea of treating the specification as an inductive hypothesis is well known in the deductive verification community where the goal is to solve the following problem: $\forall\sigma.\ P(\phi_0, \sigma)$. However, in our case, we want to apply this idea during the inductive synthesis step of CEGIS where the goal is to solve $\exists\phi.\ P(\phi, \sigma_0)$ which has not been explored before.

Definition 1 (Inductive Decomposition). *Suppose the specification is of the form* $interp_s(e) = interp_d(trans(e))$ *where* $trans$ *is the function that needs to be synthesized. Let* $trans(e')$ *be a recursive call within* $trans(e)$ *where* e' *is strictly smaller term than* e*. Inductive Decomposition is defined as the following substitution: 1. Replace* $trans(e')$ *with a special expression* $\boxed{e'}$*. 2. When inlining function calls, apply the following rules for the evaluation of* $\boxed{e'}$*:*

$$interp_d(\boxed{e'}) \longrightarrow interp_s(e')$$
$$\boxed{e'} \text{ in any other context} \longrightarrow trans(e')$$

i.e. Inductive Decomposition works by delaying the evaluation of a recursive $trans(e')$ call by replacing it with a placeholder that tracks the input e'. Then, if the algorithm encounters these placeholders when inlining $interp_d$ in the specification, it replaces them directly with $interp_s(e')$ which we know how to evaluate, thus, eliminating the need to inline the unknown $trans$ function. This replacement is sound because the specification states $interp_d(trans(e')) = interp_s(e')$. If the algorithm encounters the placeholders in any other context where the inductive specification can not be leveraged, it defaults to evaluating $trans(e')$.

Theorem 1. *Inductive Decomposition is sound and complete. In other words, if the specification is valid before the substitution, then it will be valid after the substitution and vice-versa.*

A proof of this theorem can be found in the tech report [6]. Although the Inductive Decomposition algorithm imposes restrictions on which recursive calls can be eliminated, it turns out that for many of the ADT transformation scenarios, the algorithm can totally eliminate all recursive calls to *trans*. For instance, in the running example, because of the inductive structure of dstInterpret , all placeholders for recursive desugar calls will occur only in the context of dstInterpret (desugar(e')) which can be replaced by srcInterpret (e') according to the algorithm. Thus, after the substitution, the desugar function is no longer recursive and moreover, the desugaring for the different variants can be synthesized separately. For the running example, we gain a 20X speedup using this optimization. Our system also implements several generalizations of the aforementioned optimization that are detailed in the tech report [6].

4 Evaluation

Benchmarks. We evaluated our approach on 23 benchmarks as shown in Fig. 6. All benchmarks along with the synthesized solutions can be found in the tech report [6]. Since there is no standard benchmark suite for morphism problems, we chose our benchmarks from common assignment problems (the lambda calculus ones), desugaring passes from SKETCH compiler and some standard data structure manipulations on trees and lists. The AST optimization benchmarks are from a system that synthesizes simplification rules for SMT solvers [17].

Templates. The templates for all our benchmarks use one of the four generic descriptions we have in the library. All benchmarks except arrAssertions , NegNorm and AST optimizations use a generalized version of the recursiveReplacer generator seen in Fig. 3 (the exact generator is in the tech report). This generator is also used as a template for problems that are very different from the desugaring benchmarks such as the list and the tree manipulation problems, illustrating how generic and reusable the templates can be. The arrAssertions benchmark differs slightly from the others as its ADT definitions have arrays of recursive fields and hence, we have a version of the recursive replacer that also recursively iterates over these arrays. The NegNorm benchmark requires a template that has nested pattern matching. Another interesting example of reusability of templates is the AST optimization benchmarks. All 5 benchmarks in this category are synthesized from a single library function. The template column in Fig. 6 shows the number of lines used in the template for each benchmark. Most benchmarks have a single line that calls the appropriate library description similar to the example in Sect. 2. Some benchmarks also specify additional components such as helper functions that are required for the transformation. Note that these additional components will also be required for other systems such as Leon and Synquid.

4.1 Experiments

Methodology. All experiments were run on a machine with forty 2.4 GHz Intel Xeon processors and 96 GB RAM. We ran each experiment 10 times and report the median.

Hypothesis 1: Synthesis of Complex Routines is Possible. Figure 6 shows the running times for all our benchmarks (T−opt column). SyntRec can synthesize all but one benchmark very efficiently when run on a single core using less than 1 GB memory—19 out of 23 benchmarks take ≤1 min. Many of these benchmarks are beyond what can be synthesized by other tools like Leon, Rosette, and others and yet, SyntRec can synthesize them just from very general templates. For instance, the lcB and lcP benchmarks are automatically discovering the Church encodings for boolean operations and pairs, respectively. The tc benchmark synthesizes an algorithm to produce type constraints for lambda calculus ASTs to be used to do type inference. The output of this algorithm is a conjunction of type equality constraints which is produced by traversing the AST. Several other desugaring benchmarks have specifications that involve complicated interpreters that keep track of state, for example. Some of these specifications are even undecidable and yet, SyntRec can synthesize these benchmarks (up to bounded correctness guarantees). The figure also shows the size of the synthesized solution (code column)[3].

There is one benchmark (langState) that cannot be solved by SyntRec using a single core. Even in this case, SyntRec can synthesize the desugaring for 6 out of 7 variants in less than a minute. The unresolved variant requires generating expression terms that are very deep which exponentially increases the search space. Luckily, our solver is able to leverage multiple cores using the random concretization technique [7] to search the space of possible programs in parallel. The column T−parallel in Fig. 6 shows the running times for all benchmarks when run on 16 cores. SyntRec can now synthesize all variants of the langState benchmark in about 9 min.

The results discussed so far are obtained for optimal search parameters for each of the benchmarks. We also run an experiment to randomly search for these parameters using the parallel search technique with 16 cores and report the results in the T−search column. Although these times are higher than when using the optimal parameters for each benchmark (T−parallel column), the difference is not huge for most benchmarks.

Hypothesis 2: The Inductive Decomposition Improves the Scalability. In this experiment, we run each benchmark with the *Inductive Decomposition* optimization disabled and the results are shown in Fig. 6 (T−unopt column). This experiment is run on a single core. First of all, the technique is not applicable for the AST optimization benchmarks because the functions to be synthesized are not recursive. Second, for three benchmarks—the λ-calculus ones and the

[3] Solution size is measured as the number of nodes in the AST representation of the solution.

	Bench	Description	template	code	T-opt	T-parallel	T-search	T-unopt
Desugar	lang	Running example	1	50	7.5	8.6	85.9	152.5
	langState	Running example with mutable state	1	62	⊥	527.2	1746.9	⊥
	regex	Desugaring regular expressions	1	22	2.0	3.3	9.1	3.3
	elimBo ol	Boolean operations to if else	1	21	1.5	2.9	7.5	2.4
	compAssign	Eliminates compound assignments	1	42	16.6	20.9	31.8	176.2
	langLarge	Desugaring a large language	1	126	61.2	58.0	49.7	⊥
	arrAssertions	Add out of bounds assertions	3	40	37.2	50.5	66.7	53.0
	NegNorm	Computes negation normal form	3	57	21.2	13.6	64.4	⊥
	lcB	Boolean operations to λ-calculus	1	55	43.1	47.4	40.6	47.4
	lcP	Pairs to λ-calculus	1	41	163.6	258.2	288.3	258.2
Analysis	tc	Type constraints for λ-calculus	8	41	168.9	68.0	201.9	68.0
AST optim	andLt	AST optimization 1	1	15	3.1	3.1	13.2	N/A
	andNot	AST optimization 2	1	6	2.6	3.0	13.0	N/A
	andOr	AST optimization 3	1	12	3.7	3.1	14.0	N/A
	plusEq	AST optimization 4	1	18	3.3	3.0	14.0	N/A
	mux	AST optimization 5	1	6	2.4	3.0	12.4	N/A
List	lIns	List insertion	1	12	1.5	2.3	2.2	2.1
	lDel	List deletion	2	14	4.0	4.6	4.1	3.1
	lUnion	Union of two lists	1	10	8.7	2.7	4.8	2.1
Tree	tIns	Binary search tree insertion	1	48	20.7	14.5	41.6	11.6
	tDel	Binary search tree deletion	4	63	224.8	227.4	286.1	298.9
	tDelMin	Binary search tree delete min	2	18	27.1	32.2	57.7	24.9
	tDelMax	Binary search tree delete max	2	18	25.9	30.8	54.4	25.9

Fig. 6. Benchmarks. All reported times are in seconds. ⊥ stands for timeout (>45 min) and N/A stands for not applicable.

`tc` benchmark, we noticed that their specifications do not have the inductive structure and hence, the optimization never gets triggered.

But for the other benchmarks, it can be seen that *inductive decomposition* leads to a substantial speed-up on the bigger benchmarks. Three benchmarks time out (>45 min) and we found that `langState` times out even when run in parallel. In addition, without the optimization, all the different variants need to be synthesized together and hence, it is not possible to get partial solutions. The other benchmarks show an average speedup of 2X with two benchmarks having a speedup >10X. We found that for benchmarks that have very few variants, such as the list and the tree benchmarks, both versions perform almost similarly.

To evaluate how the performance depends on the number of variants in the initial AST, we considered the `langLarge` benchmark that synthesizes a desugaring for a source language with 15 variants into a destination language with just 4 variants. We started the benchmark with 3 variants in the source language while incrementally adding the additional variants and measured the run times both with the optimization enabled and disabled. The graph of run time against the number of variants is shown in Fig. 7. It can be seen that without the optimization the performance degrades very quickly and moreover, the unoptimized version times out (>45 min) when the number of variants is >11.

4.2 Comparison to Other Tools

We compared SyntRec against three tools—Leon, Synquid and Rosette that can express our benchmarks. The list and the tree benchmarks are the typical

benchmarks that Leon and Synquid can solve and they are faster than us on these benchmarks. However, this difference is mostly due to SyntRec's final verification time. For these benchmarks, our verification is not at the state of the art because we use a very naive library for the set related functions used in their specifications. We also found that Leon and Synquid can synthesize some of our easy desugaring benchmarks that requires constructing relatively small ADTs like elimBool and regex in almost the same time as us. However, Leon and Synquid were not able to solve the harder desugaring problems including the running example. We should also note that this comparison is not totally apples-to-apples as Leon and Synquid are more automated than SyntRec.

For comparison against Rosette, we should first note that since Rosette is also a SyGus solver, we had to write very verbose templates for each benchmark. But even then, we found that Rosette cannot get past the compilation stage because the solver gets bogged down by the large number of recursive calls requiring expansion. For the other smaller benchmarks that were able to get to the synthesis stage, we found that Rosette is either comparable or slower than SyntRec. For example, the benchmark elimBool takes about 2 min in Rosette compared to 2 s in SyntRec. We attribute these differences to the different solver level choices made by Rosette and Sketch (which we used to built SyntRec upon).

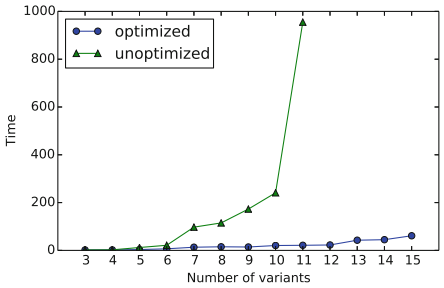


Fig. 7. Run time (in seconds) versus the number of variants of the source language for the langLarge benchmark with and without the optimization.

5 Related Work

There are many recent systems that synthesize recursive functions on algebraic data-types. Leon [3,8,9] and Synquid [16] are two systems that are very close to ours. Leon, developed by the LARA group at EPFL, is built on prior work on complete functional synthesis by the same group [10] and moreover, their recent work on Synthesis Modulo Recursive Functions [8] demonstrated a sound technique to synthesize provably correct recursive functions involving algebraic data types. Unlike our system, which relies on bounded checking to establish the correctness of candidates, their procedure is capable of synthesizing provably correct implementations. The tradeoff is the scalability of the system; Leon supports using arbitrary recursive predicates in the specification, but in practice it is limited by what is feasible to prove automatically. Verifying something like equivalence of lambda interpreters fully automatically is prohibitively expensive, which puts some of our benchmarks beyond the scope of their system.

Synquid [16], on the other hand, uses refinement types as a form of specification to efficiently synthesize programs. Like our system, Synquid also depends on bi-directional type checking to effectively prune the search space. But like Leon, it is also limited to decidable specifications. There has also been a lot of recent work on programming by example systems for synthesizing recursive programs [1,4,13,14]. All of these systems rely on explicit search with some systems like [13] using bi-directional typing to prune the search space and other systems like [1] using specialized data-structures to efficiently represent the space of implementations. However, they are limited to programming-by-example settings, and cannot handle our benchmarks, especially the desugaring ones.

Our work builds on a lot of previous work on SAT/SMT based synthesis from templates. Our implementation itself is built on top of the open source Sketch synthesis system [18]. However, several other solver-based synthesizers have been reported in the literature, such as Brahma [5]. More recently, the work on the solver aided language Rosette [20,21] has shown how to embed synthesis capabilities in a rich dynamic language and then how to leverage these features to produce synthesis-enabled embedded DSLs in the language. Rosette is a very expressive language and in principle can express all the benchmarks in our paper. However, Rosette is a dynamic language and lacks static type information, so in order to get the benefits of the high-level synthesis constructs presented in this paper, it would be necessary to re-implement all the machinery in this paper as an embedded DSL.

There is also some related work in the context of using polymorphism to enable re-usability in programming. [11] is one such approach where the authors describe a design pattern in Haskell that allows programmers to express the boilerplate code required for traversing recursive data structures in a reusable manner. This paper, on the other hand, focuses on supporting reusable templates in the context of synthesis which has not been explored before. Finally, the work on hole driven development [12] is also related in the way it uses types to gain information about the structure of the missing code. The key difference is that existing systems like Agda lack the kind of symbolic search capabilities present in our system, which allow it to search among the exponentially large set of expressions with the right structure for one that satisfies a deep semantic property like equivalence with respect to an interpreter.

6 Conclusion

The paper has shown that by combining type information from algebraic datatypes together with the novel Inductive Decomposition optimization, it is possible to efficiently synthesize complex functions based on pattern matching from very general templates, including desugaring functions for lambda calculus that implement non-trivial Church encodings.

Acknowledgments. We would like to thank the authors of Leon and Rosette for their help in comparing against their systems and the reviewers for their feedback. This research was supported by NSF award #1139056 (ExCAPE).

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_67
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 1–8 (2013)
3. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA 2013, p. 1:1–1:10. ACM, New York (2013)
4. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 229–239 (2015)
5. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI, pp. 62–73 (2011)
6. Inala, J.P., Qiu, X., Lerner, B., Solar-Lezama, A.: Type assisted synthesis of recursive transformers on algebraic data types (2015). CoRR, abs/1507.05527
7. Jeon, J., Qiu, X., Solar-Lezama, A., Foster, J.S.: Adaptive concretization for parallel program synthesis. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 377–394. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21668-3_22
8. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA, pp. 407–426 (2013)
9. Kuncak, V.: Verifying and synthesizing software with recursive functions. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 11–25. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43948-7_2
10. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 316–329 (2010)
11. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Not. **38**, 26–37 (2003)
12. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04652-0_5
13. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 619–630 (2015)
14. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. In: PLDI, p. 43 (2014)
15. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1), 1–44 (2000)
16. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 522–538. ACM, New York (2016)

17. Singh, R., Solar-Lezama, A.: Swapper: a framework for automatic generation of formula simplifiers based on conditional rewrite rules. In: Formal Methods in Computer-Aided Design (2016)
18. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis, EECS Department, UC Berkeley (2008)
19. Solar-Lezama, A.: Open source sketch synthesizer (2012)
20. Torlak, E., Bodík, R.: Growing solver-aided languages with Rosette. In: Onward!, pp. 135–152 (2013)
21. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: PLDI, p. 54 (2014)