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 `{jinala, polikarn, asolar}@csail.mit.edu`
[2] Purdue University `xkqiu@purdue.edu`
[3] Northeastern University `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 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 [7,1,12,4,15]. 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 [17] and Rosette [19,20] 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 Section 5 that all our 22 diverse benchmarks are synthesized using just 3 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 SyntRec, which is built on top of the open source Sketch synthesis platform [18]. 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 [17] and Rosette [20,19], 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 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 SyntRec. 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 Figure 1: the type `srcAST` has five *variants* (two of which are recursive), while `dstAST` has only three. In particular, the source

```
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{} GtOp{}}
```

Fig. 1: ADTs for two small expression languages

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 Section 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, Figure 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 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[†]. The task of the synthesizer is to fill in every choice and hole in such a way that the resulting program satisfies the specification.

[†] 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 GtOp());
    return new BinaryD(op = op, a= a, b = b);}
}
```

Fig. 2: Template for desugar in Sketch

The template in Figure 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)) 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 Figure 3. When the user invokes recursiveReplacer (src ,desugar), the body of the generator is specialized to the surrounding context, resulting in a template very similar to the one in Figure 2. Unlike the template in Figure 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 the synthesizer has to consider, since all the additional choices are resolved during type inference. Figure 3 also shows a compacted

```
 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[??],  field (src )));
 7 }}}
 8 generator T rcons<T>(fun e) {
 9   if (??) return e();
10   else return new cons?(rcons(e));
11 }
12 generator T field<T,S>(S e) {
13   return (e.fields?) [??];
14 }
```

```
 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], b = a[2]);
 9     case BetweenS:
10       dstAST[3] a = {desugar(src.a), desugar(src.b),
11                      desugar(src.c)};
12       return new BinaryD(op = new AndOp(),
13         a = new BinaryD(op = new LtOp(), a = a[0], b = a[1])
14         b = new BinaryD(op = new LtOp(), a = a[1], b = a[2]));
15 }}
```

Fig. 3: Left: Generic template for recursiveReplacer Right: Solution to the running example

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 Figure 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 Figure 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 10. 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** GtOp()). 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 Figure 3 are two such components.

The field component expands into an arbitrary field of type τ, where τ is derived from the context. To implement field, we use the *field list PSC* to obtain the array of all fields of type τ, and then access 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).

The rcons component is a polymorphic version of the recursive constructor for dstAST in Figure 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 10) is specialized based on the type required by that constructor and can make different non-deterministic 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. Our entire library of components can be found in the appendix. The next section gives a formal account of the syntax and semantics of SYNTREC, while Section 4 describes an important optimization that improves the scalability of constraint-based synthesis in the context of recursive ADT transformations.

3 Language

Figure 4 shows a simple kernel language that captures the relevant features of SYNTREC but elides many of the features that are orthogonal to synthesis. In this language, a program consists of a set of ADT declarations followed by a set of function declarations. The language distinguishes between a regular function f and a generator $\hat{f}$; the latter is specialized to its calling context. Expressions in the language include standard operations on ADTs such as pattern matching, field access and constructors, as well as the new *PSC*s, demonstrated in the running example. 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. Also, note that ADTs in SYNTREC are not polymorphic.

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

Fig. 4: Kernel language

3.1 Expansion Rules

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. The typing rules for the non-synthesis constructs in the language are standard and can be found in the appendix. 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 order words, after this expansion, the typing judgement $\Gamma \vdash e' : \theta$ must hold. We are not the first to note that bi-directional typing [14] can be very useful in pruning the search space for synthesis [12,15], 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.

The expansion rules for functions and *PSC*s are shown in Figure 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 is 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 of the 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), unknown constructor is exapnded into a Sketch hole by the rule UC2.

$$FUN \quad \frac{\Gamma; \{x_i : \theta_i\}_{i<n} \vdash e \xrightarrow{\theta_o} e'}{\Gamma \vdash \theta_o\ f \langle\{T_i\}\rangle \left(\{x_i : \theta_i\}_{i<n}\right) e \xrightarrow{\perp} \theta_o\ f \langle\{T_i\}\rangle (\{x_i : \theta_i\})\ e'}$$

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

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

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

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

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

Fig. 5: Expansion rules for various language constructs

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 can then be expanded with our improved knowledge of the types, and the body of the generator can then be 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 fails 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 Figure 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.

4 Synthesis

The expansion rules from Section 3 reduce the synthesis problem to a fixed number of discrete choices. Similar to SKETCH, we use a constraint-based approach to solve for these choices.

4.1 Constraint-based synthesis

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 a standard counter-example guided inductive synthesis (CEGIS) approach to solve this doubly quantified problem [17]. 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 work for a small sets 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 to the synthesis problems that arise in our context, however, poses scalability challenges due to the highly recursive nature of the function to be synthesized and the functions used in the specification. For instance, consider the example from Section 2. Since `desugar` is a recursive function, a given concrete value for the input σ such as `BetweenS(a = NumS(...), b = BinaryS(...), ...)`, exercises 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`. This greatly hinders scalability especially when the source language has many variants.

4.2 Inductive Decomposition

Inductive Decomposition solves the above problem by leveraging the inductive specification and eliminating the recursive calls to `desugar`. 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. At a high-level, this optimization works by delaying the evaluation of a recursive `desugar(e')` call by replacing it with a placeholder that tracks the input e'. Then, when evaluating `dstInterpret` in the specification, we will encounter these placeholders, but because of the inductive structure of `dstInterpret`, the placeholders will occur only in the context of `dstInterpret (desugar(e'))`. At this point, we can leverage the specification to replace `dstInterpret (desugar(e'))` directly with `srcInterpret (e')` which we know how to evaluate, thus, eliminating the need to call `desugar` recursively. Thus, 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.

Note that this optimization is not specific to this one example. Below, we first define the structural constraints necessary for the optimization and then give a formal definition of the optimization.

Definition 1 (Recursive Transformer).
*Given an ADT $\tau = \sum_i Q_i \left\{l^i_j : \tau^i_j\right\}_{j<n_i}$, a function f is a **Recursive Transformer** if it has the following form:*

$$f\,(e) := match\,(e)\,\{case\ Q_i : proc^{Q_i}(\{f(e.l^i_{j_k})\}_{k<b_i}, \{e.l^i_{j'_k}\}_{k<b'_i})\}$$

Where none of the fields $e.l^i_{j'_k}$ are of type τ (but all the fields $e.l^i_{j_k}$ must be for the recursive call to be well typed).

In other words, f will pattern match on e, and in each case it will make recursive calls on certain fields of e and process the results through an arbitrary function $proc^{Q_i}$ before returning them.

Definition 2 (Recursive Morphism).
*A **Recursive Morphism** is a **Recursive Transformer** with the following additional constraint:*

$proc^{Q_i}(v_0, \ldots v_k, v'_0, \ldots, v'_k) = \psi[v_0, \ldots v_k]$ where $v_0 \ldots v_k$ are the terms involving the recursive calls to this morphism function and ψ is an expression that constructs an ADT tree (potentially of a different type than τ) with $v_0 \ldots v_k$ as some of the leaves and ψ itself does not depend on them, so $proc^{Q_i}(u_0, \ldots u_k,\ v'_0, \ldots, v'_k) = \psi[u_0, \ldots u_k]$ with the same ψ.

For example, the desugar function is a recursive morphism where ψ for each case is the unknown expression tree of type dstAST that is generated by the rcons function in the template (line 6 in Figure 3). On the other hand, the interpreters in the running example that compute the value of an expression from the values of its sub-expressions are examples of recursive transformers by the above definition, but are not morphisms because they read the recursively evaluated values.

With these definitions in place, we can now define the Inductive Decomposition optimization.

Definition 3 (Inductive Decomposition).
*Let $intrp_s$, $intrp_d$ be two **recursive transformers** that operate on two recursive data types $\tau_s = \sum_i K_i \left\{l^i_j : \tau^{s^i}_j\right\}_{j<n_i}$ and $\tau_d = \sum_i Q_i \left\{l^i_j : \tau^{d^i}_j\right\}_{j<m_i}$ respectively and they both produce values of type τ_r. Additionally, suppose trans is a **recursive morphism** from τ_s to τ_d. Inductive Decomposition is defined as the following substitution: First, extend the destination type to $\tau'_d = \tau_d + Q^*\{args : \tau_s\}$. Then, modify the trans function as follows:*

$$trans'\,(e) := match(e)\{\ \ case\ K_i : proc^{K_i}(\{Q^*(args = e.l^i_{j_k})\}_{k<b_i}, \{e.l^i_{j'_k}\}_{k<b'_i})\}$$

Finally, extend $intrp_d$ with a new case:

$$intrp'_d\,(e) :=\ match\,(e)\,\{\ \ldots\ case\ Q^* : intrp_s(e.args)\}$$

In other words, the substitution adds an extra case to τ_d that has a single field $args$ of type τ_s. Then, the optimization changes $trans$ into $trans'$ that is no longer recursive; in every place where the function used to have a recursive call to $trans$, it now constructs a new Q^* and initializes its field $args$ with the original argument it would have passed to the recursive call. Finally, it changes $interp_d$ such that if e matches on the new variant Q^*, then instead of a recursive call to $intrp_d$ the function issues a recursive call to $intrp_s$ with the arguments that were stored in $args$.

Theorem 1. *Inductive Decomposition preserves the validity of the following constraint.*

$$intrp_s\,(e) = intrp_d(trans\,(e)) \tag{4.1}$$

In other words, if the constraint is valid before the substitution, then it will be valid after the substitution and vice-versa.

A proof of the theorem can be found in the appendix. Our system also implements several generalizations of the aforementioned optimization that are detailed in the appendix. The main limitation of the overall approach is that it is not applicable when $intrp_d$ processes the fields of its input before calling itself on the fields. A typical example of this kind of interpreter is the λ-calculus interpreter as the expression e in the application $(\lambda x.e)y$ is first modified by substituting x with y before invoking the interpreter on it. Nevertheless, we found that this optimization is applicable for many of the scenarios in our benchmark suite and leads to a significant speedup for large transformation problems.

5 Evaluation

Benchmarks We evaluated our approach on 22 benchmarks as shown in Figure 6. All benchmarks along with the synthesized solutions can be found in the appendix. 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 [16].

Templates The templates for all our benchmarks use one of the three generic descriptions we have in the library. All benchmarks except arrAssertions and AST optimizations use a generalized version of the recursiveReplacer generator seen in Figure 3 (the exact generator is in the appendix). 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. 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 Figure 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 Section 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.

5.1 Experiments

Methodology All experiments were run on a machine with forty 2.4 GHz Intel Xeon processors and 96GB 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 1GB memory—18 out of 22 benchmarks take $\leq$ 1 minute. 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, 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)[‡].

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 blows up the search space. Luckily, our solver is able to leverage multiple cores using the random concretization technique [6] to search the space of possible programs in parallel. The column T−parallel in Figure 6 shows the running times for all benchmarks when run on 16 cores. SyntRec can now synthesize all variants of langState benchmark in about 9 minutes.

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

[‡] 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	$\perp$	527.2	1746.9	$\perp$
	regex	Desugaring regular expressions	1	22	2.0	3.3	9.1	3.3
	elimBool	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	$\perp$
	arrAssertions	Add out of bounds assertions	3	40	37.2	50.5	66.7	53.0
	lcB	Boolean operations to λ-calculus	1	55	43.1	47.4	40.6	N/A
	lcP	Pairs to λ-calculus	1	41	163.6	258.2	288.3	N/A
Analysis	tc	Type constraints for λ-calculus	8	41	168.9	68.0	201.9	N/A
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. $\perp$ stands for timeout ($>$ 45 min) and N/A stands for not applicable.

the optimal parameters for each benchmark (T−parallel column), the difference is not huge for most benchmarks.

Hypothesis 2: The Inductive Decomposition optimization improves the scalability. In this experiment, we run each benchmark with the *Inductive Decomposition* optimization disabled and the results are shown in Figure 6 (T−unopt column). This experiment is run on a single core. First of all, the optimization is not applicable for some benchmarks. The templates for the AST optimization benchmarks are not recursive and hence, the optimization is not required. Out of the other 17 benchmarks, this optimization is not applicable for only 3 benchmarks—the λ-calculus ones and the tc benchmark because their specifications do not have the inductive structure required by the optimization.

But for the other benchmarks, it can be seen that *inductive decomposition* leads to a substantial speed-up on the bigger benchmarks. Two benchmarks time out ($>$ 45 minutes) 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 and incrementally added 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 Figure 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.

5.2 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.

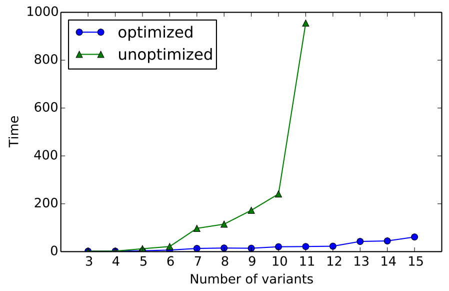

Fig. 7: Run time (in seconds) versus the number of variants of the source language for the langLarge benchmark with and without the optimization.

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 and according to the Rosette authors, 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 minutes in Rosette compared to 2s in SyntRec. These differences might be due to the different solver level choices made by Rosette and Sketch (which we used to built SyntRec upon).

6 Related Work

There are many recent systems that synthesize recursive functions on algebraic data-types. Leon [3,7,8] and Synquid [15] 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 [9] and moreover, their recent work on Synthesis Modulo Recursive Functions [7] 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 [15], 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 [12,4,1,13]. All of these systems rely on explicit search with some systems like [12] 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 [17]. 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 [20,19] 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 from Section 3 as an embedded DSL.

There is also some related work in the context of using polymorphism to enable re-usability in programming. [10] 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 [11] 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.

References

1. A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.
2. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.
3. R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.
4. J. K. Feser, S. Chaudhuri, and I. Dillig. 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, June 15-17, 2015*, pages 229–239, 2015.
5. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
6. J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive concretization for parallel program synthesis. In *International Conference on Computer Aided Verification*, pages 377–394. Springer, 2015.
7. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426, 2013.
8. V. Kuncak. Verifying and synthesizing software with recursive functions - (invited contribution). In *ICALP (1)*, pages 11–25, 2014.
9. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 316–329, 2010.
10. R. Lämmel and S. P. Jones. *Scrap your boilerplate: a practical design pattern for generic programming*, volume 38. ACM, 2003.
11. U. Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.
12. P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.
13. D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, page 43, 2014.
14. B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
15. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM.
16. R. Singh and A. Solar-Lezama. Swapper: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *Formal Methods in Computer-Aided Design*, 2016.
17. A. Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS Dept., UC Berkeley, 2008.
18. A. Solar-Lezama. Open source sketch synthesizer. 2012.

19. E. Torlak and R. Bodík. Growing solver-aided languages with rosette. In *Onward!*, pages 135–152, 2013.
20. E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54, 2014.

A Benchmarks and Library Components

All the benchmarks along with corresponding synthesized solutions can be found at https://bitbucket.org/jeevana_priya/syntrec-benchmarks/src. The library generators and components are in the lib.skh file in the above repository. The solutions generated by SyntRec are a little verbose because of the temporary variables and also beacuse the function outputs are converted into reference parameters.

B Static Semantics of SyntRec language

The typing rules for the language work as one would expect. For example, the type of a field access from a record is the type of the field.

$$\frac{\Gamma \vdash e : \{\{l_i : \tau_i\}_i\} \qquad l = l_i \qquad \tau = \tau_i}{\Gamma \vdash e.l : \tau}$$

Values in the ADT are created through constructors.

$$\frac{\tau_{adt} = \Sigma name_i \left\{\{l^i_k : \tau^i_k\}_{k<n_i}\right\} \quad name = name_t \qquad \{l_k = l^t_k \qquad \Gamma \vdash e_k : \tau^t_k\}_{k<n_t}}{\Gamma \vdash new\ name(l_j = e_j) : \tau_{adt}}$$

The most interesting rule is the one for *switch*. The rule, shown below, assumes that the argument x to *switch* is a variable whose type is an ADT where each variant corresponds to one of the cases in the switch. The body of each case is then type checked under the assumption that the type of x is the type associated with the corresponding variant.

$$\frac{\begin{array}{c}\Gamma = (\Gamma^{'}; x : \tau_{adt}) \qquad \tau_{adt} = \Sigma name_i \left\{\{l^i_k : \tau^i_k\}_{k<n_i}\right\} \\ \{(\Gamma^{'}; x : \left\{\{l^i_k : \tau^i_k\}_{k<n_i}\right\}) \vdash \ e_i : \tau\}_i\end{array}}{\Gamma \ \vdash \ switch\,(x)\,\{\ case\ name_i : e_i \ \ \}_i : \tau}$$

C Dynamic Semantics of SyntRec language

The dynamic semantics evaluate expressions under an environment σ that tracks the values of variables. ADT values are represented with a named record $\langle name,\ \{l_i = v_i\}_i\rangle$ that has the name of the corresponding variant and the values for each field. This is illustrated by the rule for the constructor.

$$\frac{\{\sigma, e_i \to v_i\}_i \qquad v = \langle name,\ \{l_i = v_i\}_i\rangle}{\sigma, new\ name\,(\{l_i = e_i\}_i) \to v}$$

The name stored as part of the record is used by the switch statement in order to choose which branch to evaluate.

$$\frac{\sigma\,(x) = \langle name,\ \{l_i = v_i\}_i\rangle \quad name_j = name \qquad \sigma, e_j \to v}{\sigma, switch\,(x)\,\{\ case\ name_i : e_i\}_i \to v}$$

and it is easy to show that the typing rules ensure that the field access rule below will always find a matching field $l_i = l$.

$$\frac{\sigma, e \to \langle name,\ \{l_i = v_i\}_i\rangle \quad l = l_i \quad v = v_i}{\sigma,\ e.l \to v}$$

D Proof of Inductive Decomposition theorem

The proof has two parts; first we show that the substitution is complete, meaning that if the original specification is valid for a given synthesized *trans* function, then it will still be valid after we perform the substitution.

Completeness: First, it should be clear that the transformation of τ_d into τ'_d and the corresponding change to $intrp_d$ by themselves will not have any effect on the validity of the specification. It is the replacement of the recursive calls to $trans(e.l^i_{j_k})$ with the constructor $Q^*(args = e.l^i_{j_k})$ that really needs to be scrutinized. In order to show that replacing all recursive calls to *trans* with the corresponding constructor will have no effect on the validity of the formula, it suffices to show that replacing one call will have no effect, because if replacing one call has no effect, then the calls can be replaced one by one until all the calls have been replaced. We start by assuming that the specification is indeed valid. Now, since *trans* is a morphism we can assume that the result of $trans\,(e)$ is a recursive ADT value $\psi[\nu_0, \ldots, \nu_n]$ where the ν_k are the result of the recursive calls $trans(e.l^i_{j_k})$ that we intend to replace, and $\psi[\circ]$ is the context in which that value appears. Because *trans* is a morphism, the context $\psi[\circ]$ is independent of the ν_k, so replacing them with some other value will not affect the context $\psi[\circ]$. This means that after the substitution, the result of *trans* will now be $\psi[\nu'_0, \ldots, \nu'_n]$ where $\nu'_k = Q^*(args = e.l^i_{j_k})$.

Now, because of the structure of the interpreter, calling $intrp_d(\psi\,[\nu_0, \ldots, \nu_n])$ will be equivalent to $intrp'_d(\psi\,[\nu'_0, \ldots, \nu'_n])$ if $intrp_d\,(\nu_k) = intrp'_d(\nu'_k)$. This last equality will be true because $intrp_d\,(\nu_k) = intrp_d\,\big(trans\,\big(e.l^i_{j_k}\big)\big)$ which equals $intrp_s(e.l^i_{j_k})$ by our assumption of the validity of the spec. Also, by the definition of the Q^* case in $intrp'_d$, it is easy to see that $intrp'_d(\nu'_k)$ will also equal $intrp_s\,\big(e.l^i_{j_k}\big)$.

Soundness: For soundness, we need to show that if the original specification does not hold for a given *trans*, then it will also not hold after the substitution. Let us assume that the specification does not hold, and let e_x be the smallest ADT value such that $intrp_s\,(e_x) \neq intrp_d(trans\,(e_x))$. We define smallest in terms of the maximum depth of recursion of e_x (the height of the tree, if we think of e_x as a tree). So we need to show that there exists an e'_x such that $intrp_s\,(e'_x) \neq intrp'_d(trans'\,(e'_x))$. Now, if e_x is not recursive, then $e'_x = e_x$ and we are done, since the substitution will only affect recursive values. Now, if e_x is recursive, then $e_x = K_i(\{l^i_j = e^i_j\}_{j<n_i})$ for some constructor K_i , but because e_x is the smallest ADT value for which the spec does not hold, then $intrp_s\,\big(e^i_j\big) = intrp_d(trans\,\big(e^i_j\big))$ holds for all e^i_j and all their sub-values, which means $intrp'_d\,(trans'\,(e_x)) = intrp_d\,(trans\,(e_x)) \neq intrp_s(e_x)$, so also $e'_x = e_x$.

Note that the proof of soundness above only works because the recursive calls inside *trans* operate on trees that are smaller than the input tree. This is an important condition that is implied by the definition of a Morphism; if it were not to hold, the transformation could take a buggy implementation (one that has an infinite recursion, for example) and make it appear correct.

E Generalizations of Inductive Decomposition

Inductive decomposition can be generalized relatively easily to cases when the interpreter takes additional arguments. For example, the definition of recursive transformers can be extended to include functions that take additional parameters and thread them through the computation possibly after changing them.

$$f\,(e,\ S) := match\,(e)\,\{case\ Q_i : proc^{Q_i}(\{f(e.l^i_{j_k},\ S'_{ik})\}_{k<b_i}, \{e.l^i_{j'_k}\}_{k<b'_i})\}$$

This is useful, for example, for an interpreter that must take as input the state of the program in addition to an AST. In this case, the specification will have the form

$$intrp_s\,(e,\ S) = intrp_d\,(trans\,(e)\,,\ S)$$

The substitution of *trans* does not need to account for these additional parameters, but the substitution of $intrp_d$ will now have to pass the additional parameters to the new call to $intrp_s$.

$$intrp'_d\,(e, S) :=\ match\,(e)\,\{\ \ldots\ case\ Q^* : intrp_s(e.args, S)\}$$

This works because we are replacing what would have been a call to $intrp_d(\nu, S)$ where ν would have been the result of calling $trans(e.args)$ with a call to $interp_s(e.args, S)$.

A more interesting generalization involves the case when $trans(e)$ takes some parameters. For example, in cases when the transformation is type directed, *trans* may take as a parameter a symbol table which gets updated as part of the recursive calls to *trans* . In that case, the specification will look like the one below.

$$intrp_s\,(e,\ S) = intrp_d\,(trans\,(e, \Gamma)\,,\ S)$$
$$\text{assuming}\ \ \Gamma = F(S);$$

Without the constraint that Γ is a function of S, the specification would almost guarantee that *trans* ignores Γ, since Γ only appears on the right-hand side of the equality. When *trans* is a type directed transformation, for example, F would produce a symbol table with the types of the variables in the state S.

Just like $intrp_s$ in the previous case, the *trans* function must thread its parameter Γ through its recursive calls, possibly transforming it along the way. The functions $proc^{K_i}$ will be able to examine this gamma when deciding what term to construct.

In order to cope with this case, the inductive decomposition will have to add an additional field to Q^* to track this parameter.

$$trans'\,(e, \Gamma) := match(e)\{case\ K_i : proc^{K_i}(\{Q^*(args = e.l^i_{j_k}, targs = \Gamma_{ik})\}_{k<b_i}, \{e.l^i_{j'_k}\}_{k<b'_i})\}$$

The transformation of $interp_d$ will look much like it did in the previous generalization, but in addition to adding a call to to $interp_s(e.arg, S)$ for the Q^* case, the optimization will also have to include an assertion that $e.targs = F(S)$. This additional assertion guarantees soundness, but it sacrifices completeness because it imposes an additional requirement regarding the relationship between $interp_d$ and $trans$. Namely, that the relationship that $\Gamma = F(S)$ should hold not just on the inputs, but also during all recursive steps. For simple interpreters and morphisms, this is not a significant constraint; it will hold whenever the state of the transformer is some abstraction of the program state (as is the case with types). The cases where it does not hold, are cases where the state Γ tracks some aspect of the program that is not tracked by the interpreter, for example, whether a given construct has been seen or not. In such cases, however, it is relatively easy to extend the interpreter to track this additional aspect as part of its state.