Type Assisted Synthesis of Recursive Transformers on Algebraic Data Types

Jeevana Priya Inala  
MIT  
jinala@mit.edu

Xiaokang Qiu  
MIT  
xkqiu@csail.mit.edu

Ben Lerner  
Brown  
blerner@cs.brown.edu

Armando Solar-Lezama  
MIT  
asolar@csail.mit.edu

Abstract

In this paper, we show how synthesis can help implement interesting functions involving pattern matching and algebraic data types. One of the novel aspects of this work is the combination of type inference and counterexample-guided inductive synthesis (CEGIS) in order to support very high-level notations for describing the space of possible implementations that the synthesizer should consider. The paper also describes a set of optimizations that significantly improve the performance and scalability of the system.

The approach is evaluated on a set of case studies which most notably include synthesizing desugaring functions for lambda calculus that force the synthesizer to discover Church encodings for pairs and boolean operations, as well as a procedure to generate constraints for type inference.

1. Introduction

Syntax-guided-synthesis [2] has gained recent popularity as an approach to synthesis that leverages a syntactic description of the space of possible programs to improve the scalability of synthesis. Systems that follow this approach can be categorized by whether the space of programs is built into the tool, or whether the system provides a flexible mechanism to describe the spaces of programs, as is the case in Sketch [25], Rosette [31, 32] or the many SyGuS solvers [3], with some systems taking the middle ground of providing a high-level notation to describe some limited aspect of the space (i.e. the looping structure as in [23]).

Recent work on synthesis of recursive transformations on algebraic data-types (ADTs) [11][10][15][22] has favored the built-in-space approach, both for ease-of-use and for efficiency reasons. Having the space of programs built in spares the programmer the error-prone process of describing a space of programs, and it has the added advantage of allowing domain-specific synthesis algorithms to exploit the structure of the space in order to gain efficiency. There is, however, a strong case for flexible systems that give the programmer more control and can help even with problems that are too difficult to be synthesized completely automatically but that can be synthesized with some user guidance.

This paper attacks the problem of recursive transformations on ADTs from the perspective of a general system with full flexibility over the spaces of programs. First, we introduce a new set of type-directed synthesis constructs that enable us to describe program spaces in a highly reusable manner. This means that for most problems, a user does not have to write her own description but can rely instead on a generic description from a library. Even when the user does write her own description, the new constructs can help reduce the scope of error and make the synthesis process more efficient thanks to the way they leverage type information to prune the search space. The paper also describes a new optimization called inductive decomposition to aid the synthesis of recursive morphisms on ADTs. We show that by using this optimization, we can achieve asymptotic improvements in the complexity of synthesizing morphisms over ADTs. The efficiency gained by this optimization, together with the ability of our solver to leverage multiple cores, allows our system to attack problems that could not be synthesized otherwise.

We implemented all these ideas in a tool called SyntRec which is implemented as an extension to the open source Sketch synthesis system [26]. Our tool supports very general correctness specifications that can use arbitrary functions to constrain behavioral properties. Like other general systems such as Sketch [25] or Rosette [31, 32]—and unlike more specialized systems such as Leon [15]—our system relies on exhaustive bounded checking to establish correctness. While this does not provide strong correctness guarantees, it works well in practice and allows us to tackle problems that would otherwise be undecidable and are beyond the scope of any other systems. For example, some of our benchmarks involve desugaring functions, where the behavioral constraint is defined in terms of the output of an interpreter on the original and desugared ASTs. We show, for example, that our system can automatically infer Church encodings for pairs and booleans from a lambda calculus interpreter. In another case-study, we show that the system is powerful enough to infer type constraints for a simple language from a description of the semantics of the type constraints. Moreover, several of our benchmarks come from transformation passes implemented in our own compiler and synthesizer, and we are now incorporating simplification rules synthesized by this system into our own synthesizer. However, synthesizing desugaring passes or other compilation transformations for larger languages is beyond the scope of the current work; nevertheless, the work in this paper is an important step in that direction.

2. Overview

In order to describe the synthesis features in the language, we use the problem of desugaring a simple language as a running example. Specifically, the goal is to synthesize a function

```plaintext
dstAST desugar(srcAST src){ ... }
```
write a single line of code in the body of the desugar function. This space is described through a sketch and reusable way while still achieving significant scalability benefits. The language makes it possible to define these spaces in a very concise details unspecified. Restricting the space of choices from which the scriptions from which the srcAST is shown below.

```java
generator T recursiveReplacer <: T, Q>(Q src, fun rec)
switch src{
    case T[] a = map(src, T rec);
    return ??(a, src ??);
}

The generator above describes a very general computational pattern. When the generator is instantiated in a particular context with specific types for T and Q, the repeat_case is expanded into pattern matching code that matches on the exact type of src. The body of each case will recursively call the function on all the fields of src that are of type T, and for each case, the resulting function will return a new sub-tree whose leaves will either be constants or the results of the recursive calls or the fields of src. Our system takes about 20s to synthesize this code using 16 cores and the final code generated from the one line sketch is shown in Figure 2. It is worth noting that while the generator itself is not trivial to write, the same generator can be used to synthesize desugaring functions for a variety of languages, because while the details of the desugar function will be different for different languages, those details will be discovered by the synthesizer in every case; the generator only describes the common high-level structure.

Supporting efficient synthesis from a highly reusable generator like the one above requires a new approach to leveraging type information and combining it with existing constraint-based synthesis techniques. It also requires new class of optimizations that leverage the structure of desugaring functions in order to allow the synthesizer to reason about these functions in a modular way, significantly boosting efficiency and scalability. The next section provides a high-level overview of the language and the main type-based mechanisms used to generate a low-level sketch from a high-level generator like recursiveReplacer.

### 3. Language

The SyntRec language is built on top of Sketch, which is a relatively large language that includes support for arrays, high-order functions, closures, and a number of other features that are not really relevant to this work. In order to make explanations clearer and formalization more tractable, we start by defining a simple kernel language with no synthesis constructs (shown in Figure 3) and then extend it with the new synthesis constructs. The kernel language captures the relevant features of SyntRec but elides many of the features that are orthogonal to synthesis with algebraic data types. In the kernel language, a program consists of a set of ADT declarations followed by a set of function declarations. Also, following standard practice, the kernel language elides the distinction between expressions and statements, so for example, we assume that the body of a function is an expression.

#### 3.1 Static Semantics

As is customary, we formalize ADTs as tagged unions \( \tau = \sum_{\text{variant}_i} \), where each of the variants is a record type \( \text{variant}_i = \)

\[ \text{let } a = \text{srcInterpret}(\exp) \]
\[ \text{assume } a != \text{null} \]
\[ \text{assert}(a === \text{dstInterpret}(\text{desugar}(\exp))) \]

The constraint above states that interpreting an arbitrary expression \( \exp \) in the source language should be equivalent to desugaring \( \exp \) and interpreting the resulting expression under the destination language. The functions srcInterpret and dstInterpret are two interpreters written in SyntRec and defined recursively over the structure of the respective ADTs, although it is worth noting that for cases where the destination language is a subset of the original language, one could reuse the first interpreter in writing the second one. As we will explain in Section 4, our synthesizer contains a novel optimization called inductive decomposition that can take advantage of the structure of the above specification in order to significantly improve the scalability of the synthesis process. Note, all ADT values in SyntRec are immutable and the operator === recursively compares two different ADT values.

The programmer must also describe a space of possible implementations from which the desugar function must be synthesized. This space is described through a sketch, a partial program that describes the high-level structure of the solution while leaving the details unspecified. Restricting the space of choices from which the synthesizer can choose from has been shown to be an effective way to make synthesis tractable, but the need to write detailed sketches has been a major obstacle for the usability of many synthesis techniques. Thanks to the new constructs presented in this paper, however, our language makes it possible to define these spaces in a very concise and reusable way while still achieving significant scalability benefits comparable to what one can achieve with very specialized and hard to write sketches.

In the case of the above example, the programmer only has to write a single line of code in the body of the desugar function.

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

3\( ^1 \) The code in the figure was cleaned up manually for readability, for example by removing temporary variables that appear in the actual generated code. The actual outputs from our synthesizer can be found in the supplementary material. It is also worth noting that while the synthesizer is biased to produce small expressions, there is no guarantee of minimality for the generated code.
The base is expanded under the set of expressions, it is straightforward to convert expressions containing the new constructs to sets of expressions in the kernel language of Figure 3. Once the new constructs have been expanded into sets of expressions, it is straightforward to convert them to programs with unknown constants as has been described in previous work \cite{27}.

As the example of the unknown field $e$.?? illustrated, the expansion into sets of expressions requires propagating type information both top-down and bottom up. Our expansion rules achieve this by using bi-directional rules that make this flow of information explicit \cite{24}. We are not the first to note that bi-directional typing can be very useful in pruning the search space \cite{22}, 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 rules have the form shown below; the transformation is parameterized by a set of types $T = \{\tau_0 ... \tau_n\}$, and the result of applying the transformation is a set of expressions $\{e_0 ... e_k\}$.

$$T = \{\tau_0 ... \tau_n\}$$

$$\Gamma \vdash e \mapsto \{e_0 ... e_k\}$$

The transformation guarantees that the type of each expression $e : \tau \in \{e_0 ... e_k\}$ belongs to the set $T$. By transforming the expression $e$ under a set of types $T$, we propagate top down information about the type of expressions required in a given context.

**Unknown fields** Below is the rule for expanding the unknown fields shown earlier.

$$\Gamma \vdash e.?? \mapsto \{e_{i,l_j} | e_{i,l_j} : \tau \text{ i.e. } i \in [0,k] \text{ and } \tau \in T\}$$

The base is expanded under the set $T'$ of all types that have fields of the correct type. As we will show shortly, if $e$ is just a variable, the corresponding variant and the values for each field. This is illustrated by the rule for the constructor.

$$\sigma, e_i \rightarrow v_i \quad v = \langle \text{name, } l_i = v_i \rangle$$

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

$$\sigma(x) = \langle \text{name, } l_i = v_i \rangle \quad \text{name} = \text{name} \quad \sigma, e_j \rightarrow v$$

$$\sigma, \text{switch(x)} \{ \text{case name} : e_j \} \rightarrow v$$

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

$$\sigma, e \rightarrow \langle \text{name, } l_i = v_i \rangle \quad l = l_i \quad v = v_i$$

$$\sigma, e.l \rightarrow v$$

Overall, the static and dynamic semantics are fairly standard, but they will be important to understand the new synthesis constructs.
resulting set \( \{e_0 \ldots e_k\} \) will just be a singleton if the type of \( e \) is in \( T' \).

**Fields List** Fields list \( e.(\tau) \) is another synthesis construct that is also used in the running example and it returns an array of all fields of \( e \) that have the type \( \tau \). There are two rules for this construct depending on whether the set \( T \) contains \( \tau \) i.e. an array of type \( \tau \).

\[
\begin{align*}
\tau_0 & \notin T \\
\Gamma \vdash e.(\tau_0) & \rightarrow \{\} \\
\end{align*}
\]

where \( \tau_0 \) is a singleton; \( \{\} \) is a singleton set.

**repeat_case** As in the running example, a single \texttt{repeat_case} is expanded into multiple cases; for each \texttt{case} \( \texttt{repeat_case} \) is expanded setting \( x \) to a different type corresponding to each variant \( \texttt{name}\{i\} \) of \( \tau \).

\[
\begin{align*}
\Gamma & = (T'; x : \tau_1 \ldots \tau_k) \\
\Gamma \vdash e.(\tau) & \rightarrow E_i = \{e_0 \ldots e_k\}
\end{align*}
\]

For each variant \( i \), the body of \( \texttt{repeat_case} \) is expanded setting \( x \) to a different type corresponding to each variant \( \texttt{name}\{i\} \) of \( \tau \).

\[
\begin{align*}
\Gamma & = \{\texttt{switch}(x) \{\texttt{repeat_case} : e\}\} \\
\end{align*}
\]

The rule only contemplates the case when \( T = \{\tau\} \) is a singleton; it is easy to generalize the rule to the case when \( T \) is a bigger set—by transforming under each element of the set and taking the union of the resulting expression sets—but this generalization is not necessary in practice since all switch statements are of type Unit in the full-fledged language.

**Unknown constructors** The language supports the creation of objects with unknown type. For each algebraic data-type in \( T \) and for each constructor of that ADT, we identify a set of fields \( J_i \) that match the fields passed to the unknown constructor. Since we know the types of those fields, we can transform their values \( e_i \) under a precise type and use \texttt{choice} to initialize the fields in the relevant constructor.

\[
\forall \tau = \Sigma \texttt{name}\{i\} \{l_i^1 : \tau_1 \ldots l_i^k : \tau_k\} \in T
\]

for each \( i \) let \( J_i \) be the maximal set of fields \( s.t. \).

\[
\forall j \in J_i \exists t, \ l_t = l_j \ and \ \Gamma \vdash e.(\tau) \rightarrow E_{j}\)
\]

\[
\Gamma \vdash \texttt{new} \{i \}
\]

We use the function \texttt{choice}(\( E \)) to make it explicit that we are asking the synthesizer to choose a single expression out of a set. We do this by defining \texttt{choice} in terms of integer unknowns which the underlying SKETCH language knows how to resolve. If \( E \) is a singleton \( \{e\} \), then \texttt{choice}(\( E \)) = \( e \). Otherwise, we can partition \( E = E_1 \cup E_2 \) into two non-empty and non-overlapping sets and define \texttt{choice} recursively as \texttt{choice}(\( E \)) = \{ \texttt{choice}(\( E_1 \)) \} else \texttt{choice}(\( E_2 \)).
This means that $e_1$ can be transformed under the singleton: $\{\tau\}$. 

$$
\Gamma \vdash e_1 \xrightarrow{\tau} E_1 \\
\Gamma; x : \tau \vdash e_2 \xrightarrow{T} \{e_2 \ldots e_{\vec{k}}\} \\
\Gamma \vdash \textbf{let} x : \tau = e_1 \textbf{ in } e_2 \\
\xrightarrow{T} \\
\{\textbf{let } x : \tau = \text{chose } (E_1) \textbf{ in } e_2 \}_{i \in [0, \vec{k}]}$
$$

We again rely on the fact that all the elements of $E_1$ have a unique type to introduce $\text{chose}$ and ask the solver to choose among these expressions. For functions, the type of the arguments and the return value are given by the function declaration. If the return type does not match the type required by the transformation, the call must be transformed to the empty set.

$$
\tau_0 \ f(in : \tau_m)\{e_b\} \\
\Gamma \vdash e \xrightarrow{\{\tau_{\text{mon}}\}} E \\
\tau_0 \in T \\
\tau_0 \notin T \\
\Gamma \vdash f(e) \xrightarrow{T} \{f(\text{chose } (E))\} \\
\Gamma \vdash f(e) \xrightarrow{T} \emptyset
$$

**Example 3.1.** Consider the program below together with its type definitions:

$$
\tau_A = \{l_1 : \tau_A \ l_2 : \tau_B \ l_3 : \tau_{\text{int}}\} \\
\tau_B = \{l_2 : \tau_A \ l_3 : \tau_{\text{int}}\} \\
\tau_{\text{int}} = \text{foo } (x : \tau_{\text{int}}) \{\textbf{let } y : \tau_{\text{int}} = x.??\?? \text{ in } y + 1\}
$$

Our transformation rule for let requires us to evaluate

$$
\Gamma \vdash x.??\?? \xrightarrow{(\tau_{\text{mon}})} E
$$

since $\tau_{\text{mon}}$ is the declared value of $y$. Both $\tau_A$ and $\tau_B$ have fields of type $\tau_{\text{int}}$, so the rule requires the transformation

$$
\Gamma \vdash x.??\?? \xrightarrow{(\tau_A, \tau_B)} E'
$$

Again, since both types have fields of type $\tau_A$ and $\tau_B$, the transformation requires us to transform $x$ under the set $\{\tau_A, \tau_B\}$ which results in the transformation below:

$$
\Gamma \vdash x \xrightarrow{(\tau_A, \tau_B)} \{x\}
$$

This means that $E'$ will be equal to $\{x.l_1, x.l_2\}$ and thus

$$
\Gamma \vdash x.??\?? \xrightarrow{(\tau_{\text{mon}})} \{x.l_1.l_3, x.l_2.l_4, x.l_3.l_4\}
$$

Therefore, let $y : \tau_{\text{int}} = x.??\??$ will be desugared into

$$
\textbf{let } y : \tau_{\text{int}} = \text{chose } \{x.l_1.l_3, x.l_2.l_4, x.l_3.l_4\}
$$

which in turn will be equivalent to

$$
\textbf{let } y : \tau_{\text{int}} = \text{if } ?? \text{ then } (\text{if } ?? \text{ then } x.l_1.l_3 \text{ else } x.l_2.l_4) \text{ else } x.l_2.l_4.
$$

### 4.4 Synthesis

The transformation rules from Section 3 allow us to reduce the synthesis problem to a problem of synthesizing integer unknowns. Solutions to this problem have been described for simple imperative programs, but solving for these integer unknowns in the context of algebraic data-types and highly recursive programs poses some new challenges that have not been addressed by prior work.

### 4.1 Background

The constraint-based approach to synthesis is to reduce the problem to one of solving a doubly quantified constraint of the form

$$
\exists \phi. \forall \sigma. Q(\phi, \sigma)
$$

where $\phi$ is a control vector describing the values of all the unknown integer and boolean constants, $\sigma$ is the input state of the program, and $Q(\phi, \sigma)$ is a predicate that is true if the program satisfies its correctness requirements under input $\sigma$ and control vector $\phi$. In general, the space of possible input states can be unbounded, but it is common to focus only on bounded spaces of inputs.

Our system follows the standard approach of unrolling loops and inlining recursive calls to derive $Q$ and uses counterexample guided inductive synthesis (CEGIS) to search for the control vector. The basic idea behind CEGIS is to construct a set of representative inputs $E$ such that solutions that work for all inputs in $E$ are likely to work for all inputs [27]. The set $E$ is initialized to a random value, and then the candidate $\phi$ derived from it is checked to ensure that it works for all inputs. If a counterexample is found, it is added to the set $E$ and the process is repeated.

Applying the standard approach to the synthesis problems that arise in our context, however, poses significant scalability challenges.

In the sections that follow, we describe a number of new optimizations that take advantage of the structure of ADT transformers in order to produce significant efficiency and scalability improvements.

### 4.2 Inductive Decomposition

Suppose that the specification imposes the following constraint:

$$
intrp_s(e) = intrp_d(trans(e))
$$

(4.1)

Where $intrp_s$ operates on a source ADT $\tau_s = \sum_{i=1}^{\lambda} K_i \{l_i \rightarrow \tau_i\}$ and $intrp_d$ operates on a destination ADT $\tau_d = \sum_{i=1}^{\lambda} \{l_i \rightarrow \tau_i\}$, and they both return some result type $\tau_r$. Now, assume the function $intrp_d$ has the following form:

$$
intrp_d(e) := \text{match } e \{\text{case } Q : \text{process}^{\phi}_{\delta}(\{\text{intrp}_d(e.l)\})\}
$$

In other words, $intrp_d$ will pattern match on the type of $e$, and in each case it will make recursive calls on certain fields of $e$ and process the results through an arbitrary function we call $\text{process}_{\delta}$ which we define later. Here, $\text{process}_{\delta}$ function is also assumed to not directly use $e.l_i$ in its computation except through recursive calls to $intrp_d$ i.e. the fields of $e$ can only flow to the output through recursive calls to $intrp_d$. Additionally, suppose $trans(e)$ is a morphism of the following form:

$$
trans(e) := \text{match } e \{\text{case } K : \text{constr}^{K}(\{\text{trans}(e.l)\})\}
$$

The transformation function $trans$ has a similar structure to $intrp_d$; it also pattern matches on the type of $e$, and in each case it also performs recursive calls on fields of $e$, but instead of processing the result, it will simply construct some new values that include the results of $\text{trans}(e.l_j)$. When all of these conditions are satisfied, we can perform an optimization we call inductive decomposition that is summarized by the theorem below. It is worth noting that we use the names $intrp_s$ and $trans$ because the optimization was inspired by transformations of ASTs like the one from the overview where the transformation must preserve behavior over two interpreters written in SYNTREC. However, the transformation is more general, and applies to problems that are not interpreters as we show later in the experiments.

**Theorem 4.1.** Given a set of functions like $intrp_s$, $intrp_d$ and $trans$ above related by a specification like the one in Equation (4.1), the following substitution preserves the validity of the specification. In other words, if the specification is valid before the substitution, then it will be valid after the substitution and vice-versa. The substitution works as follows. First, extend the destination type to $\tau_d = \tau_s + \{\text{args} : \tau_s\}$ in other words, we are adding an extra case to $\tau_d$ that has a single field $\text{args}$ of type $\tau_s$. Then, extend $intrp_d$ with a new case:

$$
intrp_d(e) := \text{match } e \{\ldots \text{case } \tau_d : intrp_d(e.args)\}
$$
i.e. if \( e \) matches on the new variant \( Q^* \), then instead of a recursive call to \( \text{intrp}_d \) we issue a recursive call to \( \text{intrp}_p \) with the arguments that were stored in \( \text{args} \). Finally, we modify the \( \text{trans} \) function as follows:

\[
\text{trans}'(e) := \text{match}(e)\{ \\
\text{case } K : \text{constr}^K(\{Q^*(\text{args} = e.i)\}) \}
\]

In other words, the new \( \text{trans}' \) is no longer recursive; in every place where we used to have a recursive call to \( \text{trans} \), we instead construct a new \( Q^* \) and initialize its field \( \text{args} \) with the original argument we would have passed to the recursive call to \( \text{trans} \).

Before outlining the proof of the theorem above, it is important to explain what is gained with the described transformation. The most important thing to keep in mind is that we are performing this transformation in the context of a synthesis problem where \( \text{constr}^K \) is actually an unknown piece of code which the synthesizer is trying to discover. Moreover, the synthesizer uses a counterexample guided inductive synthesis algorithm, where at each iteration of the algorithm, there is a concrete value \( e \) and the synthesizer is adding constraints to enforce that the specification is satisfied for that concrete value \( e \). Now under the original specification, a given concrete \( e \) will exercise multiple cases within \( \text{trans} \). This means that when the synthesizer is solving for the different \( \text{constr}^K \) it has to jointly search for all the \( \text{constr}^K \) at once, because all the constraints involve multiple \( \text{constr}^K \) together. By contrast, after the transformation, \( \text{trans}' \) is not recursive, so each counterexample will exercise only one case. This allows the synthesizer to reason about each \( \text{constr}^K \) independently, allowing it to scale to ADTs with very large numbers of cases. In Section 6 we show that whereas the original problem scaled exponentially in the number of cases in \( \text{trans} \), after the transformation, this will no longer be the case.

**Proof** 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 \( \text{trans} \) function, then it will still be valid after we perform the substitution.

**Completeness:** First, it should be clear that the transformation of \( \tau_d \) into \( \tau_d' \) and the corresponding change to \( \text{intrp}_d \) by themselves will not have any effect on the validity of the specification. It is the replacement of the recursive calls to \( \text{trans}(e.l) \) with the constructor \( Q^*(\text{args} = e.i) \) that really needs to be scrutinized. In order to show that replacing all recursive calls to \( \text{trans} \) 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, suppose that the result of \( \text{trans}(e) \) is a recursive ADT value \( \psi[\nu] \) where \( \nu \) is the result of the call \( \text{trans}(e.l) \) that we intend to replace, and \( \psi'[\nu] \) is the context in which that value appears. Note that because of the structure of \( \text{trans}(e) \), the context \( \psi'[\nu] \) is independent of \( \nu \), so replacing \( \nu \) with some other value will not affect the context \( \psi'[\nu] \). This means that after the substitution, the result of \( \text{trans} \) will now be \( \psi'[\nu'] \) where \( \nu' = Q^*(\text{args} = e.i) \). Now, because of the structure of the interpreter, calling \( \text{intrp}_d(\psi'[\nu']) \) will be equivalent to \( \text{intrp}_d(\psi[\nu']) \) if \( \text{intrp}_d(\nu) = \text{intrp}_d(\nu') \). This last equality will be true because \( \text{intrp}_d(\nu) = \text{intrp}_d(\text{trans}(e.l)) \) which equals \( \text{intrp}_d(e.i) \) by our assumption of the validity of the spec. Also, by the definition of the \( Q^* \) case in \( \text{intrp}_d \), it is easy to see that \( \text{intrp}_d(\nu') \) will also equal \( \text{intrp}_d(e.i) \).

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

Note that the proof of soundness above only works because the recursive calls inside \( \text{trans} \) operate on trees that are smaller than the input tree. This is an important condition that needs to be enforced by the transformation because if this were not the case, the transformation could take a buggy implementation (one that has an infinite recursion, for example) and make it appear correct.

### 4.3 Generalizations

The scheme described earlier can be generalized relatively easily to cases when the interpreter takes additional arguments. For example, an interpreter may take as input the state of the program in addition to an AST. In this case, the specification will have the form

\[
\text{intrp}_d(e, S) = \text{intrp}_d(\text{trans}(e), S)
\]

Where \( S \) need not be a single parameter. In this case, the interpreter will have a similar form as before, but it can now pass an arbitrary parameter \( S \) to the recursive calls.

\[
\text{intrp}_d(e, S) := \text{match}(e)\{ \text{case } Q : \text{process}(\{\text{intrp}_d(e.i, S_q)\}) \}
\]

The substitution of \( \text{trans} \) does not need to account for these additional parameters, but the substitution of \( \text{intrp}_d \) will now have to pass the additional parameters to the new call to \( \text{intrp}_d \).

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

\[
\text{intrp}_d(e, S) = \text{intrp}_d(\text{trans}(e, \Gamma), S)
\]

Assuming \( \Gamma = F(S) \).

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

The \( \text{trans} \) function itself will have the following form.

\[
\text{trans}(e, \Gamma) := \text{match}(e)\{ \text{case } K : \text{constr}^K(\{\text{trans}(e.i, K^\Gamma)\}), \Gamma\}
\]

In this case, \( \text{constr}^K \) can examine the contents of \( \Gamma \) to determine what to construct, and the recursive calls need to be passed a new parameter \( K^\Gamma \). After applying inductive decomposition, \( \text{trans} \) will have the following form.

\[
\text{trans}(e, \Gamma) := \text{match}(e)\{ \text{case } K : \text{constr}^K(\{\{Q^*(\text{arg} = e.i, \text{targ} = K^\Gamma)\}, \Gamma\}
\]

The transformation of \( \text{intrp}_d \) will look much like it did before, but in addition to substituting the recursive call to \( \text{intrp}_d(e.i, S_q) \) with a call to \( \text{intrp}_d(e.arg, S_q) \), the optimization will also have...
to include an assertion that \( \text{e.target} = F(S_{Q_1}) \). This additional assertion guarantees soundness, but it sacrifices completeness because it imposes an additional requirement regarding the relationship between \( \text{interp}_\text{d} \) and \( \text{trans} \). 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 \( \Gamma \) 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.

We also implemented the generalization for cases where \( \text{trans} \) can take mutable parameters like global counters. In this case, we inline the recursive \( \text{trans} \) calls to evaluate the mutable parameters, but still apply the transformation for the final output. In addition, this optimization can also be easily extended to cases where we have multiple specifications that constrain the \( \text{trans} \) function and all of them have the same form as in Equation (4.1) and satisfy the properties necessary for this optimization.

Finally, in some cases, it may be desirable to write an user defined \textit{equals} function instead of using the built-in \( =\) as or \( \equiv\) operators. In order for this optimization to work in those cases, the users have to explicitly tag their \textit{equals} function as “replaceable” so that the system can assume that the two parameters of the \textit{equals} function call are equivalent. Here, it is the responsibility of the users to correctly tag functions and we do not provide any guarantees other than basic type checking.

Finally, the main limitation of this optimization is that it is not applicable when \( \text{interp}_\text{d} \) processes the fields of its input before calling itself on the fields. A typical example of this kind of interpreter is the \( \lambda \)-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.

### 4.4 Merging of Recursive Calls

The optimization described above effectively eliminates recursion from the transformer, but the interpreters can still contain large numbers of recursive functions. For example, the interpreter from the simple source language from the running example will have more than a dozen recursive calls, so a symbolic execution approach that relies on inlining these calls to even a small depth will be prohibitively expensive.

Our system addresses this problem by merging recursive calls with mutually exclusive path conditions. The basic idea is illustrated in Figure 5. This is done as a preprocessing pass before any inlining takes place: the synthesizer follows a simple heuristic to identify calls with mutually exclusive path constraints; it simply labels calls on opposite sides of conditional or on different cases of a switch statement as mutually exclusive. With the exception of our simpler benchmarks, none of our other benchmarks solve without this optimization. We recently learned that this transformation is a special case of the optimization published by Lal and Quadeer only a few weeks ago [18].

**Figure 5.** Function merging optimization.

This additional optimization descriptively eliminates recursion from the transformer, but the interpreters can still contain large numbers of recursive functions. For example, the interpreter from the simple source language from the running example will have more than a dozen recursive calls, so a symbolic execution approach that relies on inlining these calls to even a small depth will be prohibitively expensive.

Our system addresses this problem by merging recursive calls with mutually exclusive path conditions. The basic idea is illustrated in Figure 5. This is done as a preprocessing pass before any inlining takes place: the synthesizer follows a simple heuristic to identify calls with mutually exclusive path constraints; it simply labels calls on opposite sides of conditional or on different cases of a switch statement as mutually exclusive. With the exception of our simpler benchmarks, none of our other benchmarks solve without this optimization. We recently learned that this transformation is a special case of the optimization published by Lal and Quadeer only a few weeks ago [18].

<table>
<thead>
<tr>
<th>Bench</th>
<th>Recursive data types</th>
<th>ADT as objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_tlhs</td>
<td>18.42</td>
<td>21.57</td>
</tr>
<tr>
<td>sim_bet</td>
<td>60.06</td>
<td>535.16</td>
</tr>
<tr>
<td>sim_betState</td>
<td>648.39</td>
<td>TO</td>
</tr>
<tr>
<td>sim_lcB</td>
<td>517.92</td>
<td>TO</td>
</tr>
<tr>
<td>sim_lcP</td>
<td>1662.12</td>
<td>TO</td>
</tr>
<tr>
<td>sim_tc</td>
<td>496.12</td>
<td>TO</td>
</tr>
</tbody>
</table>

**Figure 6.** Run times (in seconds) comparing recursive data types approach to relational encoding for various benchmarks (TO > 2700s)

### 4.5 Random concretization and parallel search

In addition to these optimizations, we also leverage the random concretization technique of \textsc{Sketch} [14] to search the space of possible programs in parallel and we found that this greatly helps us to scale to much difficult problems. However, we do not use the technique to randomly concretize all holes in the sketch, but only specialize it to concretize the depths of \textsc{GuCs}. The main intuition for this is that only a few of \textsc{GuCs} in the sketch actually require larger depths, but most of the others require very small depths. Hence, the concretization is biased towards small values and by doing this, we also get the added advantage of prioritizing small expressions.

### 4.6 Encoding of ADT values

The \textsc{Sketch} synthesis system on which our system is built does not support algebraic data types, so extending \textsc{Sketch} to support ADTs was also an important aspect of this work. One approach to encoding ADT values is to adapt techniques developed for solver-based analysis of object oriented programs. A common family of techniques uses relations and relational operations to model the heap and then translates these relational operations into \textsc{SAT} [7][13][33]. The original sketch language uses similar techniques to encode heap allocated structures, so it is relatively straightforward to extend that encoding to model ADT values—we refer to this encoding as the relational encoding. Unfortunately, our early experiments with this encoding showed that it was not sufficiently scalable. In Figure 6 we show the result of an early experiment we conducted comparing this encoding with the encoding we describe in the next section. These experiments were run on early versions of our benchmarks that were less demanding than the final versions described in the evaluation section and before we had implemented inductive decomposition. The experiments show unambiguously that this approach would not be sufficiently scalable.

The main disadvantage of the relational encoding is that it fails to exploit the fact that ADT values are \textit{immutable} and can therefore be treated as values instead of references. Immutability allows us to aggressively apply equational reasoning to simplify the formulas before they are even converted to \textsc{SAT}. By contrast, when the heap is represented as a set of relations as it is in prior work, the solver must do extra work to discover that, for example, the initialization of the fields of a newly allocated object will not affect the field values of previously allocated objects. This is because initializing the fields of the newly allocated object involves modifying the same relation that stores the values of the fields of the previously allocated objects.

Instead of this approach, we use an encoding of immutable recursive data-structures that takes advantage of immutability and leverages equational reasoning to do aggressive simplifications of the formulas. Support for equations involving immutable recursive data-structures is now present in a number of solvers including \textsc{Z3}, \textsc{CVC3} and \textsc{Yices} [4][6][8]. However, we found that we could achieve significantly better performance by eagerly bit-blasting directly to \textsc{SAT} after algebraic simplification as described in the next section.

In Figure 7 for example, we show the results of a comparison
between Z3 and our own encoding on a set of inductive synthesis problems from the same set of simplified benchmarks that were used in Figure 6; the times reported in Figure 7 are smaller than those in Figure 6 because Figure 6 is the end-to-end synthesis time, whereas Figure 7 includes only the inductive synthesis time. Once again, these are not the same benchmarks that were used for our evaluation; these benchmarks were less demanding, but also did not use inductive decomposition.

The reason for this is that the inductive synthesis problems where the synthesizer spends the bulk of its time do not really leverage the power of the theory solvers available in a state-of-the-art SMT solver like Z3. These inductive synthesis problems have many terms involving recursive data-structures, but because all the inputs are concrete, the only true unknowns are single bits or very small integers. This means that after the formula is split into a boolean part and a theory part, once the SAT solver proposes an assignment, the theory solver can check that assignment simply by doing constant propagation; anything more sophisticated than that becomes unnecessary.

### 5. From Recursive Data-types to SAT

Our solver encodes all formulas as DAGs, where the sources are nodes corresponding to constants and either inputs or holes—depending on whether we are in the inductive synthesis or the checking phase of CEGIS—and the sinks are assertions. For the rest of the section we formalize the DAGs as lists of node definitions in three-address-code in the following notation: \( dag = [ x_i \leftarrow \psi_i ] \).

#### 5.1 The language of constraints

Our system implements 25 different types of nodes \( \psi \) to support a variety of boolean, integer and array operations, but for the sake of conciseness, we limit our presentation to the following 8 types of nodes.

\[
\psi := \begin{cases} \text{?}_i \text{id} & \text{eq} (x_0, x_1) \mid N \mid \text{Assert}(x_1) \\ \text{ite}(x_2, x_0, x_1) \mid + (x_0, x_1) \\ \text{TC}(x_1, \ldots, x_n) \mid \text{TR}(x_1, n) \end{cases}
\]

The node \( \text{TC}(x_1, \ldots, x_n) \) stands for tuple creation and creates a tuple with values represented by nodes \( x_1 \) to \( x_n \); some of the \( x_n \) values can be empty, but the DAG must have assertions to ensure that those empty values cannot flow to another assertion (this is guaranteed by the fact that the constraints come from well-typed programs). The node \( \text{TR}(x_1, n) \) stands for tuple read; the node \( x_1 \) is expected to be a tuple, and \( n \) is an integer constant that determines which field of the tuple is to be read. Reading from an empty field can be treated like a havoc value.

The translation from the SYNTREC language into this language of constraints is straightforward and is best illustrated with an example.

#### Example 5.1.

Consider a trivial sketch; we use the simplified syntax of the kernel language to make the example more concise.

\[
\begin{align*}
\text{let } x : \text{lst} &= \text{new nil}() \\
\text{let } y : \text{lst} &= \text{if } (\psi) \{ x \} \\
\text{else } \{ \text{new cons (car = \psi, cdr = x) } \} \\
\text{in switch}(y)\{ \text{case nil : assert false; } \\
\text{case cons : assert y.car == 7; } \}
\end{align*}
\]

Where the type \( \text{lst} = \text{cons \{ car : int, cdr : lst \} } + \text{nil}() \).

In order to compile this down to constraints, we first flatten the first tuple type into a tuple of type \( \text{lst} = (\text{int, int, lst}) \) where the first int is a code for whether the list is a cons (1) or is nil (0), and the other fields contain values only if the code is 1. Thus, the code above will be compiled to the following constraint (we show the integer constants inlined to save space).

\[
\begin{align*}
x_0 \leftarrow \psi \quad & x_0 \leftarrow \text{eq}(x_0, 1) \\
x_1 \leftarrow ?_1 \quad & x_1 \leftarrow \text{TR}(x_0) \\
x_2 \leftarrow TC(0, \ldots, \ldots) \\
x_3 \leftarrow TC(1, x_1, x_2) \\
x_4 \leftarrow \text{ite}(x_0, x_2, x_3) \\
x_5 \leftarrow \text{TR}(x_4)
\end{align*}
\]

The first \( \text{Assert} \) ensures that case \( \text{nil} \) is never taken, while the second \( \text{assert} \) will force \( ?_2 = 7 \).

As mentioned before, one advantage of this representation is the ability to use equational reasoning to simplify the constraints and potentially discover the value of different unknowns without even having to invoke the solver—this is the case for the example above. In the example, it is possible to discover a solution using simplification rules based on two equalities: \( TR(ite(a, b, c), n) = ite(a, TR(b, n), TR(c, n)) \) and \( TR(TC(x_0, \ldots, x_n), i) = x_i \). Using these equalities it is easy to show that \( x_2 \) will be 1 only if \( x_0 \) is \( \text{false} \); thus, the system can discover that \( ?_1 = \text{false} \) before even invoking the solver. Further simplification can then show that \( x_1 = x_3 \) and thus \( x_2 = x_1 \), allowing the two unknowns to be discovered through simplification only. In general, it is rare for formulas to simplify so dramatically, but simplification rules are very important in reducing the size of problems that must be encoded in SAT.

### 5.2 From constraints to SAT

Our encoding leverages the unary encoding used by Sketch to represent integer values \([27]\). Integer values are encoded as a list of the form \( v = [(c_i, b_i)] \), where each pair in the list is composed of a constant \( c_i \) and a SAT variable \( b_i \). The previously published encoding is made more compact by leveraging a modified version of MiniSat \([9]\) which in addition to standard CNF clauses also supports uniqueness constraints \( \text{unique}(b_0, \ldots, b_n) \) that lazily generate CNF clauses to enforce that for every unary value only one \( b_i \) is true and all other ones are false \([11]\).

A detailed description of the encoding is provided as an appendix, but the high-level idea is that every node \( TC \) in the formula is assigned a unique id. Tuple values are then represented as integers \( v = [(c_i, b_i)] \) where each \( c_i \) corresponds to a tuple id and \( v \) will be equal to that tuple if the corresponding \( b_i \) is true. When reading \( \text{TR}(v, k) \) from one of these values, the encoder finds all the relevant tuples \( c_i \) and choses among their fields \( k \) based on which \( b_i \) is true.

This can be illustrated with Example 5.1—assuming it had not been simplified. There are two tuples corresponding to \( x_2 \) and \( x_3 \) which we can assume have ids 2 and 3 respectively. So value \( x_4 \) will be an integer encoded as \( [(2, b_1), (3, b_2)] \) where \( b_1 \) will be true if \( ?_1 \) is true, and \( b_2 \) will be true if \( ?_2 \) is false. Thus, when...
reading field 0 from $x_4$, we get a value $[(0, b_1), (1, b_2)]$ because when $b_1$ is true, $x_4$ is equal to object 2 whose field 0 has value 0 and something similar happens when $b_2$ is true.

6. Evaluation

All the above ideas are implemented in a tool called SyntReC which is an extension to the open source Sketch synthesis system [20]. To demonstrate the effectiveness of these ideas, we used SyntReC to synthesize several useful and interesting recursive functions involving pattern matching from high level sketches.

Our benchmark suite consists of 21 interesting functions that include synthesizing desugarm work for language constructs, generating type constraints, optimizing abstract syntax trees and manipulating data structures like lists and trees. A link to all our benchmarks along with the synthesized solutions can be found in the appendix. Figure 8 lists these benchmarks along with their descriptions, the search space represented by their templates, and the runtimes for various experiments. We chose some of these benchmarks from standard exercises from introduction to Programming Languages classes like Church encodings and collecting type constraints. We chose some of the other benchmarks based on what we encounter in our own work like AST optimizations, adding array out of bounds assertions, etc. The list and tree manipulation benchmarks are pretty standard ones.

Each one of these benchmarks consists of a set of ADT definitions that are relevant for the synthesis problem, sketch of the function to be synthesized, and a specification that constrains the desired functionality. As in the running example, these specifications can use arbitrary functions (like interpreters) and hence, these benchmarks also contain implementations for these functions. To make it easier for the users to write the sketch, we have designed a library of template functions that can be used directly in the sketch. Most of our benchmarks use these templates, and in some cases, we specialize these templates by making some minor modifications to adapt to the problem at hand.

For all these benchmarks, we only do bounded verification. In fact to avoid the case where verification takes too much time, we bound the depth of algebraic data types in the specification to 2, integer inputs to 3 bits, inlining bound to 3, GUC depth to 3 and use minimal loop unrolling amount and array sizes that are sufficient to synthesize the correct function. All these bounds can be changed by command line flags. For the benchmarks that fail either because the inlining or the GUC depth was not sufficient, we increased these bounds until the system succeeds. The system actually prints out relevant error messages for these failures, so it is easy to know which bound to increase. Also, if the problem is underspecified because of the depth of ADTs in specification or input bits size, we increased these values until the synthesizer found the correct solution consistently. We manually verified all output solutions and the system also includes a code generator that produces a random tester that can be used to improve the confidence in the generated code.

We also have some benchmarks where the inductive decomposition optimization is not applicable. For example, λ-calculus interpreters do not have the recursive structure required by this optimization. The sketches of the AST optimizations and list insertion benchmarks are not recursive and hence, the optimization does not apply there too. The specification for the tc benchmark is complex that does not immediately reveal the inductive hypothesis required for the optimization.

We now describe the experiments done on these benchmarks to validate the hypotheses made in this paper.

6.1 Experiments

Methodology All our experiments were run on a machine with forty 2.4 GHz Intel Xeon processors and 99 GB RAM, and running Ubuntu 14.04.1, LTS. We ran each experiment ten times with a timeout of 45 minutes and report the median. All parallel experiments were run using 16 threads.

Hypothesis 1: Synthesis of complex routines is possible Most of our benchmarks are beyond the scope of what can be synthesized by other tools like Leon, Rosette, and others, and we show that SyntReC can synthesize these functions in a couple of seconds to a couple of minutes. The run time column in Figure 8 shows the running times (in seconds) for the benchmarks with all optimizations enabled and with random concretization for GUC depth holes using 16 cores. In most of these benchmarks, the synthesizer is generating non-trivial code from very general sketches. For instance, the icb benchmark is automatically discovering the Church encodings to boolean operators like And, Or and Not down to pure λ-calculus. Similarly, the benchmark ieP synthesizes translation from a language supporting pairs and operations on pairs (First and second) to simple λ-calculus. Here, the correctness for these benchmarks is defined in terms of the full fledged interpreters for the both languages. Another interesting benchmark is tc which synthesizes an algorithm to produce type constraints for lambda calculus ASTs to be used in order to do type inference. The output of this sketch is a conjunction of type equality constraints which the algorithm produces by traversing the AST. The specification for this benchmark uses a unification algorithm to simplify the output constraint and asserts equivalence between the expected type of an arbitrary expression (computed using a simple type inference algorithm) and its inferred type from the generated constraints. The betState benchmark is also interesting as it extends the running example with constructs that can mutate state and the interpreters take state as a parameter. With this extension, the desugaring of BetweenS obtained in Figure 2 is incorrect when b modifies the state since it is evaluated twice. A correct desugaring must appropriately use LetD constructs in the destination language to avoid this problem. Synthesizing this translation is trickier than the running example because this desugaring requires constructing dstAST trees of depth at least 5 and the solution space increases exponentially with depth. Similarly arr Assertions and compAsign benchmarks also use interpreters that mutate state and are more close to real world imperative language interpreters.

For list and tree benchmarks, we use a very naive library for the set related functions that are used in the specification. Because of this, the system spends most of the time just in verifying the final synthesized solution compared to the time spent on synthesis. For example, for the tns benchmark, the system only spends in about 20s in synthesis and the rest of the time is spent in verification. And similarly, the tDel benchmark takes only 150s in synthesis and takes about 300s to verify.

Hypothesis 2: Inductive decomposition optimization greatly improves performance In this experiment, we wanted to measure the benefits of the inductive decomposition optimization described in Section 4.2. The Unopt column in Figure 8 reports the running times when this optimization is disabled for those benchmarks where the optimization is applicable. This experiment is also run with random concretization for GUC depth holes using 16 cores. For some of the smaller benchmarks, the optimization does not make a big difference. But, for the other benchmarks, the optimization improves performance by 1.4X to 5X. Moreover, two of our benchmarks timeout (> 45 minutes) when the optimization is disabled. On the other hand, there are also cases where the optimization worsens performance and this is especially seen in the benchmarks involving lists and trees. On a closer look at these results, the difference in performance is because of the verification times and not the synthesis times. Thus,
The synthesis problem here is as follows: Given an abstract syntax part of a tool to develop simplification rules for our solver itself. This claim is substantiated by the fact that 8 out of 9 of our more efficient and easier to develop. Here, we use S case of using S optimization benchmarks. These benchmarks also illustrate the use for the benchmark. Even for the other benchmarks that could not use this template, their sketches are still very similar to different functions. Moreover, even for the other benchmarks that actually does bounded verification for only depth 2 ADT inputs, but for this experiment, we increased this depth to 3 so that variants are now more dependent on each other. We also reduced the number of constructs to 7 to avoid blowing up the verification times. We started the benchmark with just 3 constructs in the source language and incrementally added the additional constructs and measured the run times both with the optimization enabled and disabled. The graph of runtimes with number of variants is shown in Figure 9.

**Hypothesis 3: Reusability of sketches** Another main contribution of this work is the design of the new synthesis constructs such that it is possible to write highly general and reusable sketches. This claim is substantiated by the fact that 8 out of 9 of our desugaring benchmarks use the exact same template used in the running example. Hence, all of their sketches involve writing a single line of code and yet, all these benchmarks synthesize very different functions. Moreover, even for the other benchmarks that could not use this template, their sketches are still very similar to this template and have just a couple of lines more that are specific for the benchmark.

Another interesting example of reusability of sketches is the AST optimization benchmarks. These benchmarks also illustrate the use case of using SYNTRREC in making synthesis enabled tools both more efficient and easier to develop. Here, we use SYNTRREC as part of a tool to develop simplification rules for our solver itself. The synthesis problem here is as follows: Given an abstract syntax tree node that can be optimized, the goal is to find an optimized formula and a predicate that governs when this optimization can be applied. The specification asserts that both the optimized version and the original AST generate same outputs for all possible input assignments. In this system, a single sketch is used to generate hundreds of simplification rules and here, we report the results for five of them.

**Hypothesis 4: Impact of the new synthesis constructs and type directed reduction rules (or) comparison to normal syntax guided synthesis** To measure how much the type information from algebraic data types helps in restricting the space of possible so-

<table>
<thead>
<tr>
<th>Bench</th>
<th>Description</th>
<th>Search space</th>
<th>Run time</th>
<th>Unopt</th>
<th>Single thread</th>
<th>NonADT (single thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bet</td>
<td>Running example: Desugaring between construct</td>
<td>2^330</td>
<td>19.36</td>
<td>117.31</td>
<td>61.61</td>
<td>591.32</td>
</tr>
<tr>
<td>betState</td>
<td>Desugaring between with mutable state</td>
<td>2^1064</td>
<td>166.93</td>
<td>TO</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>regex</td>
<td>Desugaring regular expressions</td>
<td>2^140</td>
<td>2.53</td>
<td>3.57</td>
<td>1.94</td>
<td>18.88</td>
</tr>
<tr>
<td>elimBool</td>
<td>Boolean operations to if else</td>
<td>2^27</td>
<td>2.10</td>
<td>2.32</td>
<td>1.23</td>
<td>2.83</td>
</tr>
<tr>
<td>compAssign</td>
<td>Eliminates compound assignments</td>
<td>2^118</td>
<td>11.73</td>
<td>TO</td>
<td>45.94</td>
<td>204.69</td>
</tr>
<tr>
<td>mergeOp</td>
<td>Merge operations into one construct</td>
<td>2^359</td>
<td>36.96</td>
<td>84.92</td>
<td>248.07</td>
<td>TO</td>
</tr>
<tr>
<td>arrAssertions</td>
<td>Add out of bounds assertions</td>
<td>2^509</td>
<td>17.22</td>
<td>28.09</td>
<td>220.26</td>
<td>1636.94</td>
</tr>
<tr>
<td>lcB</td>
<td>Boolean operations to lambda calculus</td>
<td>2^438</td>
<td>58.15</td>
<td>N/A</td>
<td>97.06</td>
<td>TO</td>
</tr>
<tr>
<td>lcP</td>
<td>Pairs to lambda calculus</td>
<td>2^201</td>
<td>138.05</td>
<td>N/A</td>
<td>1970.04</td>
<td>TO</td>
</tr>
<tr>
<td>tc</td>
<td>Type constraints for lambda calculus</td>
<td>2^292</td>
<td>324.27</td>
<td>N/A</td>
<td>1009.13</td>
<td>TO</td>
</tr>
<tr>
<td>andLt</td>
<td>AST optimization 1</td>
<td>2^120</td>
<td>2.84</td>
<td>N/A</td>
<td>3.60</td>
<td>78.22</td>
</tr>
<tr>
<td>andNot</td>
<td>AST optimization 2</td>
<td>2^120</td>
<td>2.42</td>
<td>N/A</td>
<td>1.48</td>
<td>55.75</td>
</tr>
<tr>
<td>andOr</td>
<td>AST optimization 3</td>
<td>2^120</td>
<td>4.36</td>
<td>N/A</td>
<td>41.31</td>
<td>76.64</td>
</tr>
<tr>
<td>plusEq</td>
<td>AST optimization 4</td>
<td>2^120</td>
<td>8.97</td>
<td>N/A</td>
<td>45.12</td>
<td>66.02</td>
</tr>
<tr>
<td>mux</td>
<td>AST optimization 5</td>
<td>2^120</td>
<td>2.09</td>
<td>N/A</td>
<td>1.74</td>
<td>86.45</td>
</tr>
<tr>
<td>lIns</td>
<td>List insertion</td>
<td>2^15</td>
<td>1.79</td>
<td>N/A</td>
<td>1.46</td>
<td>1.51</td>
</tr>
<tr>
<td>lDel</td>
<td>List deletion</td>
<td>2^64</td>
<td>3.19</td>
<td>2.92</td>
<td>2.85</td>
<td>8.4</td>
</tr>
<tr>
<td>lUnion</td>
<td>Union of two lists</td>
<td>2^30</td>
<td>11.78</td>
<td>7.48</td>
<td>11.53</td>
<td>5.21</td>
</tr>
<tr>
<td>tSch</td>
<td>Search in a Binary Search tree</td>
<td>2^627</td>
<td>2.99</td>
<td>N/A</td>
<td>2.50</td>
<td>4.61</td>
</tr>
<tr>
<td>tIns</td>
<td>Binary search tree insertion</td>
<td>2^528</td>
<td>62.81</td>
<td>41.56</td>
<td>89.94</td>
<td>27.94</td>
</tr>
<tr>
<td>tDel</td>
<td>Binary search tree deletion</td>
<td>2^528</td>
<td>478.89</td>
<td>1677.59</td>
<td>707.78</td>
<td>908.79</td>
</tr>
</tbody>
</table>

**Figure 8.** Benchmarks. All reported times are in seconds. TO stands for timeout (> 45 min).

**Figure 9.** Total time (in seconds) versus number of variants of source language for the mergeOp benchmark with and without optimization. These experiments use 16 cores to do parallel search.
ADTs like 

was increased by 1, but S 

with a single thread and the benchmark 

hypothesis rules improve ease of programmability compared to normal syntax guided synthesis and yet, also achieve great performance.

**Hypothesis 5: Comparison to Rosette**  
Rosette is a solver aided language that can in principle express all our benchmarks. However, Rosette does not contain the high level synthesis constructs and hence, we had to manually desugar all the constructs. This increased the size of the sketch and in some cases, also gave the Rosette solver more information that in SYNTREC. For most of our benchmarks, 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. Note, we also gave Rosette similar inlining bounds that we used in SYNTREC and we verified the template by manually feeding it the solution that SYNTREC found. For the other smaller benchmarks that was 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 even in Rosette which is two times slower than SYNTREC. Also, Rosette does not scale when the maximum depth of ADTs to be considered was increased by 1, but SYNTREC still synthesized the code in almost the same time showing that SYNTREC is more robust than Rosette.

**Hypothesis 6: Comparison to Leon**  
Leon synthesizer is another system that can synthesize recursive functions involving algebraic data types. List and some of the tree benchmarks are the typical benchmarks that Leon can solve and Leon is faster than ours on these benchmarks. We also found that Leon can synthesize some of our desugaring benchmarks that requires constructing relatively small ADTs like elimBool and regex. But, Leon was not able to synthesize our other complicated benchmarks. Even for the simple running example, Leon could only solve the trivial cases and was not able to solve the the non-trivial Between5 case. Even here, we verified that the specifications given are correct by manually providing the right solution. The system was able to verify in just a couple of seconds. We also tried to bound both the input and output by adding a constraint on the depth in the pre and post conditions, but this didn’t help scalability.

7. Related Work

The most relevant piece of related work is the synthesizer Leon by the LARA group at EPFL [5][15][16], which builds on prior work on complete functional synthesis by the same group [17]. In particular, their recent work on Synthesis Modulo Recursive Functions [15] 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 probably 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. Leon uses a two stage process to synthesize programs: (a) Use deductive inference rules to break the problem into smaller problems, (b) Use a combination of symbolic exploration and concrete execution to solve the smaller problems. Even though, it is possible to describe the behavioral specifications like interpreters in Leon, Leon does not scale well on these kinds of benchmarks. First, the deductive rules will not significantly reduce the problem when the specification is in the form of a complex program, especially one that manipulates state like the statefull interpreter of betState. As for the symbolic search, most of our benchmarks (especially the desugaring ones) will not leverage the more sophisticated decision procedures for sets and lists available in Leon. If Leon were to do bounded inlining using the same approach we use, they would still be left with something similar to the SMT based encoding which we show to be inefficient. When symbolic reasoning is challenging, Leon can also leverage concrete execution to filter potential CEGIS programs. This is also problematic in this setting because it is difficult to find good input test candidates that can prune the space efficiently (as the inputs need to pass the complex interpreter test) and also enumerating over these huge space of candidate programs is totally infeasible.

There has been a lot of recent work on programming by example systems, some of it focusing explicitly on recursive programs. For example, the work on type and example directed program synthesis [22] exploits both type information and input-output examples to efficiently perform explicit search to synthesize functions on algebraic data types. Their work also uses bi-directional typing rules to prune the search space, but the main difference is that they only support specifications of the form of input and output examples. For the desugaring functions, writing input output examples is not very attractive as it almost requires us to know the exact desugaring. Their system also follows a strategy of “evaluate during enumeration” by pushing the examples towards the base cases of the type derivation rules and this greatly helps them in simplifying the search space and solving smaller problems at a time. However, this approach will not work when the inputs and outputs need to go through complex interpreters before they are evaluated. Additionally, they evaluate the recursive functions in the candidate solutions by using the original input and output examples and this strategy is similar to our inductive decomposition optimization and our optimization actually generalizes this technique to complex specifications. There is also similar related work by Feser, Chaudhuri and Dilig [10] which also performs explicit search and uses examples to synthesize programs with higher order combinators like map and fold. Their approach uses inductive generalization to find hypotheses about the synthesis program in a type aware manner and uses a combination of deduction...
and best first enumerative search to choose the correct hypothesis. In addition, there is also work by Albaghouthi, Gulwani and Kincaid in using an explicit search technique to synthesize recursive programs from examples 1. Their system, called Escher, uses specialized data-structures to represent the space of implementations, and applies a clever search strategy that combines forward and backward analysis. The approach also takes advantage of observational equivalence to treat as equivalent sub-programs that produce the same output for the test inputs, even if they are different. The effect of this is equivalent to partial order reduction and can significantly reduce the size of the search-space. In a similar vein, Perelman et al. have developed an approach for Test Driven Synthesis implemented in a system called LaZy 23. The approach is also based on explicit search, and is also geared towards programming-by-example problems. The key novelty is that the approach achieves efficiency by relying on the user to provide examples in increasing order of complexity, allowing the programs to be synthesized incrementally rather than in one shot. All of these projects, however, are limited to programming-by-example settings, and cannot deal with the kind of complex specifications that we use in our benchmarks.

Our work builds on a lot of previous work on SAT/SMT based synthesis from templates/sketches. Our implementation itself is built on top of the open source Sketch synthesis system 25. However, several other solver based synthesizers have been reported in the literature, such as Brahma 12. The work on proof theoretic synthesis 28 used constraint based-synthesis to infer both program fragments and invariants, making it possible to synthesize verified code. The work on path based inductive synthesis 29, showed how to make the synthesis process more scalable by focusing on a small number of paths one at a time. More recently, the work on the solver aided language Rosette 31,32 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. As our evaluation showed, however, Rosette has trouble scaling for our benchmarks as the solver gets bogged down by the large number of recursive calls and Rosette could also potentially benefit by the optimizations mentioned in the paper. Even at the solver level, Rosette makes somewhat different tradeoffs in its decisions about what to concretize and what to keep symbolic. For the kind of benchmarks we run, where you need to run recursive functions with very high branching factors over fully symbolic trees, Rosette scalability will suffer because it will “not be symbolic enough”.

There has been a lot of prior work on decision procedures for algebraic data-types. Most recently, Suter et al. developed a set of decision procedures to reason about ADTs with recursive abstraction functions that map the ADTs into values in other decidable theories 30; this work is the basis for the Leon solver described earlier. This work builds on a lot of prior work on decision procedures for ADTs. For example, Zhang, Spina and Manna developed a decision procedure to solve combinations of Presburger Arithmetic and term algebras 14, and showed how to use this procedure to model balanced trees 19. Their work, in turn, built on the work of Oppen on decision procedures for recursively defined data-structures 21. Today, several of the most popular SMT solvers support reasoning about recursive data-structures 4,6,8. In contrast to our approach, all of these approaches are primarily geared towards verification. By contrast, our approach is very efficient at model finding, and at coping with the kinds of problems that arise in inductive synthesis, where the goal is not to check whether a formula is satisfied by all possible data-structures, but rather to discover control values that will cause a formula to be satisfied for a small set of concrete data-structures.

Finally, the work on hole driven development 20 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.

8. Conclusion

The paper has shown that by combining type information from algebraic data types together with the novel inductive decomposition optimization and enumerative encodings for integers and recursive tuples, it is possible to efficiently synthesize complex functions based on pattern matching, including desugaring functions for lambda calculus that implement non-trivial church encodings.

References

A. Benchmarks

All the benchmarks along with corresponding synthesized solutions can be found at https://bitbucket.org/jeevana_priya/syntrec/src

B. Encoding from constraints to SAT

In order to generate constraints from the dag, we use an environment $\sigma$ which maps variable names to unary values. Additionally, $\sigma$ is used to maintain a map from integers to tuples. In addition to $\sigma$, the translation also maintains a constraint store $\Omega$, which stores boolean constraints as well as the uniqueness constraints mentioned earlier.

For every node $x_i \leftarrow \psi$ in the dag representation of the formula, we apply the following rule to update the state.

$$\sigma, \Omega, \psi \xrightarrow{\text{encode}} \sigma'', \Omega', v \sigma' = \sigma'' \left[ x_i \rightarrow v \right]$$

The $\xrightarrow{\text{encode}}$ rule for each type of node $\psi$ is shown in Figure 10. It is interesting to note that several rules such as constant or tuple creation do not actually introduce any constraints. Also, the $v$ parameter to $\text{mux}$ is a constant $[n, \text{true}]$, and $\text{mux}$ will not introduce any constraints; it will simply return the variable $b_i^n$ in place of $q_{0,j}$. This, combined with some simple peephole optimizations, means that even fairly complex sketches can be resolved with a relatively small number of constraints. On the negative side, though, sketches that are heavy in arithmetic can sometimes blow up in terms of memory consumption because of the inherent growth caused by the unary representation of arithmetic operations.
\( b_i \) fresh in \( n = 2^{\text{ahits}} \)
\[
\Omega' = \Omega \land \text{unique} (b_0, \ldots, b_n)
\]
\[
\sigma, \Omega ?? \xrightarrow{\text{encode}} \sigma, \Omega', [(i, b_i) \mid 0 \leq i < n]
\]
\[
\sigma, \Omega, C \xrightarrow{\text{encode}} \sigma, \Omega', [(C, \text{true})]
\]
\[
i = \text{fresh} \quad \sigma' = \sigma[i \rightarrow (\sigma(x_1), \ldots, \sigma(x_n))]
\]
\[
\sigma, \Omega, \text{TC} (x_1, \ldots, x_n) \xrightarrow{\text{encode}} \sigma', \Omega, [(i, \text{true})]
\]
\[
v, \Omega' = \text{mux}(\Omega, \sigma(x_c), \sigma(x_0), \sigma(x_1)) \xrightarrow{\text{encode}} \sigma, \Omega', v
\]
\[
\sigma (x_1) = [(e_i^1, f_i^1) \mid 0 \leq i \leq l_1] \quad \sigma (x_2) = [(e_j^2, f_j^2) \mid 0 \leq j \leq l_2]
\]
\[
c_{i,j} = e_i^1 + e_j^2
\]
\[
\Omega' = \Omega \land \text{unique} \bigl(\{q_{i,j}\} \bigr) \land 0 \leq i < l_1, 0 \leq j < l_2 \quad \bigl(\forall_{q_{i,j}} = f_i^1 \land f_j^2\bigr)
\]
\[
\sigma, \Omega, + (x_1, x_2) \xrightarrow{\text{encode}} \sigma, \Omega', [(c_{i,j}, q_{i,j})]
\]
\[
\sigma (x_t) = [(e_i^t, f_i^t) \mid 0 \leq i \leq l_t] = v_t
\]
\[
\sigma (\sigma^t) = (v_1^t, \ldots, v_l^t)
\]
\[
v, \Omega' = \text{mux}(\Omega, v_1^t, v_2^t, \ldots, v_{l_t}^t)
\]
\[
\sigma, \Omega, \text{TR} (x_t, c) \xrightarrow{\text{encode}} \sigma, \Omega', v
\]
If any \( v_i^t \) is empty, it is treated as \([(0, \text{true})]\)

For the rules for addition and \(\text{mux}\), if multiple pairs have the same value \(c_{i,j}\) or \(c_{j,i}\), we add them to the list only once and take the disjunction of all their conditions.

**Figure 10.** Encoding rules from high-level constraints to SAT