Type Assisted Synthesis of Programs with Algebraic Data Types

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 new encoding for synthesis problems involving immutable data-structures that significantly improves the scalability of the synthesizer.

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

Algebraic data-types and pattern matching are foundational concepts in programming languages, and are particularly useful in programs that perform symbolic manipulation, such as compilers or program analysis tools. Despite their convenience, however, there are many situations where functions involving ADTs and pattern matching can become large and difficult to write correctly. In this paper, we explore how synthesis can help implement non-trivial functions based on pattern matching and algebraic data-types.

The paper builds on previous work on constraint-based synthesis from program templates or sketches. The key contribution of this work is to show that by leveraging the structure provided by the type definitions of an algebraic data-type, it is possible to synthesize large and potentially complicated routines from very high-level templates. The paper also describes a new approach to encoding constraints that arise as part of the synthesis process.

Unlike recent work on synthesizing verified implementations of recursive programs on ADTs [Kneuss et al. 2013], our system does not provide strong correctness guarantees, as it relies on bounded analysis to check the correctness of the resulting implementations. On the other hand, our system is much more expressive, both in the class of functions that can be synthesized and in the class of constraints that can be imposed on the behavior of the synthesized functions. For example, some of our benchmarks involve desugaring functions, where the behavioral constraint is defined in terms of the output on an interpreter on the original and desugared ASTs. The expressiveness of the language of behavioral constraints also distinguishes this work from previous work that aims to synthesize recursive functions from input-output examples [Albarghouthi et al. 2013].

The ideas in this paper have been implemented in a new language called SYNTREC, which was implemented as an extension to the open-source Sketch synthesis system [Solar-Lezama 2012]. In order to evaluate our approach, we synthesize a number of different routines for manipulation of different types of ASTs. Among the most interesting benchmarks are a set of desugaring functions, including the desugaring of pairs and booleans down to pure lambda calculus. Specifically, the paper makes the following contributions.

- We define a new set of synthesis constructs that allow programmers to express the high-level structure of a collection of pattern matching rules while enabling the synthesizer to derive the details of each case.
- We show how a combination of type inference and constraint-based synthesis can help us derive concrete implementations from very high-level sketches.
- We define an encoding for synthesis problems involving immutable recursive structures and show that it is more efficient than a direct encoding to SMT.
- We evaluate the approach on a set of problems involving desugaring functions, data-structure manipulations, type inference and algebraic simplification of formulas.

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

```
dstAST desugar(srcAST src){ ... }
```

that can translate from a source AST to a destination AST. The ADT definitions for these two ASTs are shown in Figure 1. The type srcAST, for example, has five different variants, two of which are recursive. Like case classes in Scala, ADTs in SYNTREC require you to name the fields in each

```
adt srcAST{
  NumS{ int v; }
  PlusS{ srcAST a; srcAST b; }
  MinusS{ srcAST a; srcAST b; }
  TrueS{ }
  FalseS{ } }
adt dstAST{
  NumD{ int v; }
  BinopD{opcode op; dstAST a; dstAST b;}
  BoolD{ bit v; } }
adt opcode{ PlusOp{} MinusOp{} }
```

Figure 1. ADT for two small expression languages

variant. We will later see how this will actually help in the definition of high-level synthesis constructs.

The first step to synthesize a function is to constrain its intended behavior through a set of test harnesses. Below is an example of a test harness that constructs an expression in the source AST, using non-deterministic values passed as input, and checks that desugar produces a correct output.

```
harness void test1(int x1, int x2){
  srcAST in = new PlusS( a = new NumS(v=x1),
                         b = new NumS(v=x2) );
  dstAST out = new BinopD( op = new PlusOp(),
                           a = new NumD(v=x1),
                           b = new NumD(v=x2) );
  assert out === desugar(in);
}
```

A few features are worth noting about the test harness above. First, values of the ADT are constructed via the **new** operator in the same way one would construct objects in Java; all values of an ADT are immutable, so the fields must be assigned at creation time by passing named arguments to the constructor. For the most part, values of an ADT behave like references in Java with a few exceptions. First, reference equality (==) is not allowed between values of an ADT, but the operator === can be used to recursively compare two different values (equivalent to using .equals in Java). Immutability, together with the inability to compare references imply that they can be treated as values rather than references. We can define similar test harnesses to test desugar on other expressions to fully constrain its behavior.

Below is the sketch from which desugar is synthesized.

```
dstAST desugar(srcAST src){
  switch(src){
    repeat_case:
      return new ??( a= desugar(src.??) ,
                     b= desugar(src.??) ,
                     op = ??, v=src.??, v=??);
}   }
```

The sketch conveys the basic structure of all the different cases: call desugar recursively on some of the fields, and construct a new AST node from the results. From this sketch, the synthesizer produces the code shown in Figure 2.

```
dstAST desugar(srcAST src){
 switch(src){
  case NumS: return new NumD(v=src.v);
  case PlusS: return new BinopD(a= desugar(src.a),
                  b= desugar(src.b), op = new PlusOp() );
  case MinusS: return new BinopD(a= desugar(src.a),
                  b= desugar(src.b), op = new MinusOp() );
  case TrueS:  return new BoolD(v=1);
  case FalseS: return new BoolD(v=0);
}}
```

Figure 2. Solution to the running example

To understand what the sketch means and how the synthesizer derived the complete code from it, it is useful to first understand the semantics of the **switch** construct in SYNTREC. The **switch** statement implements a form of pattern matching. The expression passed to switch must be a variable whose type is an ADT. Each case corresponds to a variant of the algebraic data type; within the scope of each case, the type of the variable used in the switch (src in the example) is specialized to the type indicated by the case. Unlike switch statements in Java or C/C++, cases do not "fall through".

Each case in the generated code was synthesized from the single repeat_case in the sketch. The synthesizer specializes the repeat_case block to have a separate case for each variant of the algebraic data-type; any "holes" in the repeat_case can be resolved differently for each case, allowing the body to be specialized for each of the variants. Within the body of repeat_case, there are two different kinds of holes that are new to this work. First, the expression **new** ?? is a *constructor hole* which the synthesizer will specialize to a constructor for one of the variants of the ADT—exactly which one will be determined by the synthesizer. The constructor **new** ?? is given a set of arguments corresponding to the possible arguments that the different variants may need; when the synthesizer choses which variant to actually produce, all arguments that are not relevant to that variant will be dropped. The second kind of hole is a *field selector hole*, corresponding to the expression src.??. The synthesizer will use the type of src in each case and the inferred type of the expression to restrict the set of possible fields that ?? can resolve to. Additionally, the sketch also includes *value holes* ??, which are replaced by the synthesizer with suitable constants. Value holes are more powerful in our language than in previous work, because the constants can also be values in the ADT with constants assigned to their fields, so op=?? can desugar to op= **new** PlusOp(). All of these new constructs—repeat_case and the different kinds of holes—rely on type information in order to constrain the set of possible choices that the constraint-based synthesis mechanism has to search in order to derive the correct code.

The next section elaborates the semantics of the SYNTREC language and formalizes the definitions of the new synthesis constructs.

$$
\begin{array}{rcl}
P & := & adt_i \;\; f_i \\
adt & := & adt\; name \{ \;\; variant_1 \ldots variant_n \;\; \} \\
variant & := & name \{ l_1 : \tau_1 \ldots \; l_n : \tau_n \} \\
f & := & \tau_{out}\; name\, (x_i : \tau_i) \{ \;\; e_i \; \} \\
e & := & switch\, (x) \{ \, case \;\; name_i : e_i \;\; \} \\
 & & \mid \; x \; \mid \; e.l \; \mid \; new\; name(l_i = e_i) \\
 & & \mid \; let\; x : \tau = e_1 \; in \;\; e_2 \; \mid \; f(e)
\end{array}
$$

Figure 3. Core language

3. Language

This section is written in terms of a very simple kernel language (shown in Figure 3) that 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 $\tau = \Sigma variant_i$, where each of the variants is a record type $variant_i = name_i \left\{ l_1^i : \tau_1^i \ldots l_n^i : \tau_n^i \right\}$. The typing rules for the core language work as one would expect. For example, the type of a field access is the type of the field, assuming that the expression from which the field is read is a record type.

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1 \ldots l_n : \tau_n\} \qquad l = l_i \qquad \tau = \tau_i}{\Gamma \vdash e.l : \tau}$$

Values in the ADT are created through constructors.

$$\frac{\begin{array}{c}\tau_{adt} = \Sigma name_i \left\{ l_1^i : \tau_1^i \;\ldots\; l_n^i : \tau_n^i \right\} \\ name = name_t \qquad l_j = l_j^t \qquad \Gamma \vdash e_j : \tau_j^t\end{array}}{\Gamma \vdash new\; name(l_j = e_j) : \tau_{adt}}$$

We use $l_i = e_j$ as a shorthand to indicate that there are many labeled parameters $l_0 = e_0 \ldots l_k = e_k$ passed to the constructor.

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.

$$\frac{\begin{array}{c}\Gamma = (\Gamma^{'}; x : \tau_{adt}) \qquad \tau_{adt} = \Sigma name_i \left\{ l_1^i : \tau_1^i \ldots l_n^i : \tau_n^i \right\} \\ (\Gamma^{'}; x : \left\{ l_1^i : \tau_1^i \ldots \; l_n^i : \tau_n^i \;\right\}) \vdash \;\; e_i : \tau\end{array}}{\Gamma \;\vdash \; switch\, (x) \{ \, case\, name_i : e_i \;\; \} : \tau}$$

3.2 Dynamic Semantics

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

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

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

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

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

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

Overall, the static and dynamic semantics are fairly standard, but they will be important to understand the new synthesis constructs.

3.3 Synthesis Constructs

The example in Section 2 illustrated the new synthesis constructs available in the language. In order to define the semantics of the synthesis constructs, we define a type directed desugaring transformation $\xrightarrow{T}$ which reduces all the new constructs to sets of expressions in the kernel language. The goal is to have the synthesizer chose among this set of expressions for one that satisfies the constraints. This is done by using unknown boolean constants, since the semantics of programs with unknown constants have been well described in previous work [Solar-Lezama et al. 2006].

One of the challenges of implementing the desugaring transformation is that the transformation requires us to perform type inference and desugaring in tandem; this requires type information to be propagated both top-down and bottom-up. The desugaring transformation achieves this by using bi-directional rules that make this flow of information explicit [Pierce and Turner 2000].

The rules have the form shown below; the transformation is parameterized by a set of types $T = \{\tau_0 \ldots \tau_i\}$, and the result of applying the transformation is a set of expressions $\{e_0 \ldots e_k\}$. The transformation guarantees that the type of each expression $e : \tau \in \{e_0 \ldots e_k\}$ belongs to the set T.

$$\frac{T = \{\tau_0 \ldots \tau_i\}}{\Gamma \vdash e \xrightarrow{T} \{e_0 \ldots e_k\}}$$

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. The simplest rule is the

rule for transforming a variable. There are two versions of this rule, depending on whether the type of the variable is in the set T.

$$\frac{x:\tau \in \Gamma \quad \tau \in T}{\Gamma \vdash x \xrightarrow{T} \{x\}} \qquad \frac{x:\tau \in \Gamma \quad \tau \notin T}{\Gamma \vdash x \xrightarrow{T} \emptyset}$$

The rule for field access $e.l$ finds all the types $T^{'}$ with fields of the desired named and type and then uses that set of types to transform the base expression e.

$$\frac{\Gamma \vdash e \xrightarrow{T'} \{e_0 \dots e_k\} \quad where\ T^{'} = \{\tau \mid \tau\ has\ a\ field\ l : \tau_i\ and\ \tau_i \in T\}}{\Gamma \vdash e.l \xrightarrow{T} \{e_0.l \dots e_k.l\}}$$

The top level rule for functions evaluates the body of the function under the singleton containing the declared return type of the function. Transforming the body produces a set of expressions E , but we actually want to produce a single function as opposed to a set of function, so the rule uses the construct $chose\,(E)$ to ask the synthesizer to choose among the expressions in E for one that satisfies the constraints.

$$\frac{(x_i:\tau_i) \vdash e \xrightarrow{\{\tau_o\}} E \ \ where\ E = \{e_0 \dots e_k\}}{\tau_o\ name\,(x_i:\tau_i)\{\ e\ \} \xrightarrow{\emptyset} \tau_o\ name\,(x_i:\tau_i)\{chose\,(E)\}}$$

If E is a singleton $\{e\}$, then $chose\,(E) = e$. Otherwise, we can partition $E = E_1 \cup E_2$ into two non-empty and non-overlapping sets and define chose recursively as $chose\,(E) = if\ (??)\ then\ chose\,(E_1)\,else\ chose(E_2)$. The expression ?? is an unknown constant that the synthesizer must resolve. $chose$ requires all expressions in E to be of the same type, which is true here because we transformed under a set $\{\tau_o\}$.

let requires the type of its variable to be declared, so expression e_1 can be transformed under the singleton $\{\tau\}$.

$$\frac{\Gamma \vdash e_1 \xrightarrow{\{\tau\}} E_1 \quad \Gamma; x:\tau \vdash e_2 \xrightarrow{T} \{e_2^0 \dots e_2^k\}}{\Gamma \vdash let\ x:\tau = e_1\ in\ e_2 \xrightarrow{T} \left\{let\ x:\tau = chose\,(E_1)\ in\ e_2^i \;\middle|\; i \in [0,k]\right\}}$$

We again rely on the fact that all the elements of E_1 have a unique type to introduce $chose$ 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.

$$\frac{\tau_o\ f(in:\tau_{in})\{\ e_b\ \} \quad \tau_o \in T \quad \Gamma \vdash e \xrightarrow{\{\tau_{in}\}} E}{\Gamma \vdash f(e) \xrightarrow{T} \{f(chose\,(E))\}} \qquad \frac{\tau_o\ f(in:\tau_{in})\{\ e_b\ \} \quad \tau_o \notin T}{\Gamma \vdash f(e) \xrightarrow{T} \emptyset}$$

If we apply the rules presented so far to a program with no holes, we will find that all the $chose$ expressions get called with singleton values and therefore reduce to conventional expressions without any holes. The rest of this section explains the semantics of the different holes and how they interact with the rules shown so far.

repeatcase The $repeatcase$ construct formalizes the repeat_case construct used in the running example.

$$\frac{\Gamma = (\Gamma^{'}; x\ :\sum\ name_i\left\{l_1^i:\tau_1^i \dots l_n^i:\tau_n^i\right\}) \quad (\Gamma^{'}; x\ :\left\{l_1^i:\tau_1^i \dots l_n^i:\tau_n^i\right\}) \vdash e \xrightarrow{\{\tau\}} E_i = \{e_i^0 \dots e_i^{k_i}\}}{\Gamma \vdash switch\,(x)\{\ repeatcase:e\ \} \xrightarrow{\{\tau\}} \{switch\,(x)\{\ case\ name_i : chose(E_i)\}\}}$$

The body of $repeatcase$ is transformed multiple times, once for each variant $name_i\left\{l_1^i:\tau_1^i \dots l_n^i:\tau_n^i\right\}$. 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 fields The language also supports accessing unknown fields from a record as illustrated by the rule below.

$$\frac{\Gamma \vdash e \xrightarrow{T'} \{e_0 \dots e_k\} \quad where\ T^{'} = \{\tau \mid \tau\ has\ a\ field\ l:\tau_i\ and\ \tau_i \in T\}}{\Gamma \vdash e.?? \xrightarrow{T} \{e_i.l_j \mid e_i.l_j : \tau\ \ j \in [0,k]\ \ and\ \tau \in T\}}$$

The rule is similar to the one for normal field access, but in this case it needs to find all types with fields of the right type irrespective of their name.

Unknown constructors The language supports the creation of objects with unknown type as shown in the running example. 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_t under a precise type and use $choice$ to initialize the fields in the relevant constructor.

$$\frac{\forall\,\tau = \Sigma name_i\left\{l_1^i:\tau_1^i \dots l_n^i:\tau_n^i\right\}\ \in T \quad for\ each\ i\ let\ J_i\ be\ the\ maximal\ set\ of\ fields\ s.t. \quad \forall\,j \in J_i\ \exists t.\ \ l_t = l_j^i\ \ and\ \ \Gamma \vdash e_t \xrightarrow{\{\tau_j^i\}} E_j^i}{\Gamma \vdash new\ ??\,(l_t = e_t) \xrightarrow{T} \{new\ name_i\left(l_j^i = choice(E_j^i)\right)\}}$$

Generalized unknown constructors (GUC) A GUC is a more general version of the unknown constructor that automatically creates expression trees as opposed to individual values. The syntax in sketch is ??(k, $\{e_1,\dots, e_m\}$), where k is the maximum depth of the tree and $e_1, \dots, e_m$ are expressions that can be used at the leaves of the tree. In our stylized

$$\frac{k>1 \qquad \tau = \Sigma name_i \left\{ l_1^i : \tau_1^i \ldots l_n^i : \tau_n^i \right\} \quad e_1 \xrightarrow{\{\tau\}} E_1 \ \ldots \ e_m \xrightarrow{\{\tau\}} E_m \quad new\ ??\left(l_i^j = ??_{k-1}(e_1 \ldots e_m)\right) \xrightarrow{\{\tau\}} E_{rec}}{\Gamma \vdash ??_k(e_1 \ldots e_m) \xrightarrow{\{\tau\}} \{\, choice\left(E_{rec} \cup_{i\in[1,m]} E_i\right)\}}$$

$$\frac{k=1 \quad e_1 \xrightarrow{\{\tau\}} E_1 \ \ldots \ e_m \xrightarrow{\{\tau\}} E_m}{\Gamma \vdash ??_k(e_1 \ldots e_m) \xrightarrow{\{\tau\}} \{choice\left(\cup_{i\in[1,m]} E_i\right)\}}$$

$$\frac{k>1 \qquad \tau = primitive \quad e_1 \xrightarrow{\{\tau\}} E_1 \ \ldots \ e_m \xrightarrow{\{\tau\}} E_m}{\Gamma \vdash ??_k(e_1 \ldots e_m) \xrightarrow{\{\tau\}} \{choice\left(\{??_\tau\} \cup_{i\in[1,m]} E_i\right)\}}$$

Figure 4. Desugaring Rules for generalized constructor

kernel language, we use the notation $??_k(e_1 \ldots e_m)$ and we define the semantics as shown in Figure 4.

Like we did with the switch rule, we focus only on the case where $T = \{\tau\}$ is a single type, since it is easy to generalize to the case where there are multiple types. There are three different rules corresponding to one inductive case and two base cases where τ is a primitive type and where $k = 1$. In the rule for the primitive type, we use the notation $??_\tau$ to refer to a constant hole of type τ like those available in SKETCH. Finally, in our implementation, if the user simply writes ?? in a context that requires an ADT (as when setting field op in the running example), this is just desugared to ??(5,{}).

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

$$\tau_A = \{l_1 : \tau_A \ \ l_2 : \tau_B \ \ l_3 : \tau_{int}\}$$
$$\tau_B = \{l_x : \tau_A \ \ l_y : \tau_{int} \ \ l_z : \tau_{int}\}$$
$$\tau_{int} \quad foo\,(x : \tau_A)\{ \ let\ y : \tau_{int} \ = \ x.??.??\ \ in\ y + 1\ \}$$

Our transformation rule for let requires us to evaluate

$$\Gamma \vdash x.??.?? \ \xrightarrow{\{\tau_{int}\}} \ E$$

since τ_{int} is the declared value of y . Both τ_A and τ_B have fields of type τ_{int} , so the rule requires the transformation

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

Again, since both types have fields of type τ_A and τ_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

$$\Gamma \vdash x.??.?? \ \xrightarrow{\{\tau_{int}\}} \{x.l_1.l_3,\ x.l_2.l_y,\ x.l_2.l_z\ \}$$

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

$$let\ y : \tau_{int} \ = \ choice\,(\{x.l_1.l_3,\ x.l_2.l_y,\ x.l_2.l_z\ \})$$

which in turn will be equivalent to

$$let\ y : \tau_{int} = if\ ?? \quad then \quad (if\ ??\ then\ x.l_1.l_3\ else\ \ x.l_2.l_y) \quad else \quad x.l_2.l_z$$

Example 3.2. *The sketch for the running example could have been written more concisely by using a* GUC. *Specifically, for the expression in the* **return** *statement, we could have written:*

```
repeat_case: return ??(2, {src.??, desugar(src.??)})
```

The conciseness comes at the expense of increasing the set of choices available to the synthesizer. To understand why, consider how the expression above will be desugared. First, just like in the original sketch, the call to `desugar(src.??)` *will be transformed into a different set of choices for each case. For example, once* `repeat_case` *is expanded, the call* `desugar(src.??)` *under* **case** `PlusS:` *will be transformed into* `desugar(choice({src.a, src.b}))` *since* `src` *will have type* `PlusS` *under this case.*

Then, according to the transformation rule, the generalized constructor will be transformed into a choice between an unknown constructor similar to the the one in the original example or a call to `desugar(choice({src.a, src.b}))` *which was not an option in the sketch of the original example.*

4. Synthesis

The desugaring 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 ϕ is a *control vector* describing the values of all the unknown integer and boolean constants, σ 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 σ and control vector ϕ . In general, the space of possible input states can be unbounded, but it is common to focus only on bounded spaces of inputs.

```
/* before */
if (w)
    x = f(a,b);
else
    y = f(c,d);
```

```
/* after */
t = f(w?a:c, w?b:d);
if (w)
    x = t;
else
    y = t;
```

Figure 5. Function merging optimization.

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 [Solar-Lezama et al. 2006]. The set E is initialized to a random value, and then the candidate ϕ 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.

The key new challenges in this work are: encoding algebraic data-types into the constraints and coping with the high-degree of recursion present in these problems. We first describe a simple transformation we use to reduce the degree of recursion and then describe two approaches we explored to encode ADTs as constraints.

4.2 Recursion

Recursion can be problematic because for functions such as AST transformations, a single function may contain more than a dozen recursive calls to itself—even the simple example in Figure 2 will involve ten recursive calls to desugar after all the synthesis constructs have been expanded to choices controlled by scalar holes. This makes a naïve inlining approach 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 conditionals or on different cases of a switch statement as mutually exclusive. With the exception of our simpler benchmarks (that run in like 10s), none of our other benchmarks solve without this optimization.

4.3 Encoding ADT values as objects

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 SAT [Jackson and Vaziri 2000; Vaziri and Jackson 2003; Dolby et al. 2007]. 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.

The main disadvantage of this encoding is that it fails to exploit the fact that ADT values are *immutable* and can therefore be treated as values. Immutability allows us to aggressively apply equational reasoning to simplify the formulas before they are even converted to 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.

4.4 Encoding with Recursive Data Types

An alternative approach is to directly leverage the ability of many SMT solvers—such as Z3, CVC3 and Yices—to support recursive data-types [De Moura and Bjørner 2008; Dutertre and De Moura 2006; Barrett et al. 2007]. Compared with the relational approach, this approach can more easily take advantage of immutability and leverage equational reasoning to do aggressive simplifications of the formulas.

In practice, though, we found existing support for recursive data-types—at least for Z3—to be insufficiently scalable for the complexity of problems we were interested in tackling (see Section 6). We hypothesize that the reason for this is that Z3 is tuned for handling a different class of problems from the kind of problems that arise in CEGIS. Specifically, the inductive synthesis phase in CEGIS requires solvers that are fast for model finding rather than verification, and we are only interested in solving for the unknown constants in the sketch, which are typically either a single bit or a very small integer. One implication of this is that for most expressions in the program, the set of values that an expression can take is often very small.

In the following section we describe how our own solver encodes constraints involving integers and recursive data-types down to SAT problems in order to take advantage of the characteristics of the constraints derived from inductive synthesis problems.

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 ψ 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.

$$\begin{aligned}\psi \quad := \quad & ??_{id} \mid eq\,(x_0,\ x_1) \mid N \mid Assert(x_t) \\ & \mid ite\,(x_c,\ x_0,\ x_1) \mid +(x_0, x_1) \\ & \mid TC\,(x_1,\ \ldots, x_n) \mid TR\,(x_t,\ n)\end{aligned}$$

The node $TC(x_1, \ldots, x_n)$ 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 $TR(x_t, n)$ stands for *tuple read*; the node x_t 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.*

$$\begin{aligned}&let\ x : lst = new\ nil\,()\ \ in \\ &let\ y : lst = \ if\,(??)\,\{\ x\} \\ &\qquad else\,\{new\ cons\,(car = ??,\ cdr = x)\}\ \ in \\ &switch(y)\{case\ nil:\ assert\ false; \\ &\qquad\qquad case\ cons:\ assert\ y.car == 7;\ \}\end{aligned}$$

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

In order to compile this down to constraints, we first flatten the lst *type into a tuple of type* $lst = (int, int,\ lst)$ *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).*

$$\begin{aligned}x_0 &\leftarrow ??_1 & x_6 &\leftarrow eq\,(x_5, 1) \\ x_1 &\leftarrow ??_2 & x_7 &\leftarrow Assert\,(x_6) \\ x_2 &\leftarrow TC\,(0, -, -) & x_8 &\leftarrow TR\,(x_4, 1) \\ x_3 &\leftarrow TC\,(1, x_1, x_2) & x_9 &\leftarrow eq\,(x_8, 7) \\ x_4 &\leftarrow ite\,(x_0, x_2, x_3) & x_{10} &\leftarrow Assert(x_9) \\ x_5 &\leftarrow TR\,(x_4, 0)\end{aligned}$$

The first $Assert$ *ensures that case* nil *is never taken, while the second 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_5 will be 1 only if x_0 is $false$; thus, the system can discover that $??_1 = false$ before even invoking the solver. Further simplification can then show that $x_4 = x_3$ and thus $x_8 = x_1$, 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 [Solar-Lezama et al. 2006]. 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 [Eén and Sörensson 2003] which in addition to standard CNF clauses also supports uniqueness constraints $unique(b_0,\ \ldots, b_n)$ that lazily generate CNF clauses to enforce that for every unary value only one b_i is true and all other ones are false [Ganesh et al. 2012].

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 $TR(v, k)$ 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 $??_1$ 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

Our evaluation demonstrates that SYNTREC can synthesize various useful and challenging recursive functions involving pattern matching from high level sketches.

6.1 Benchmarks

We have 13 benchmarks that include synthesizing simple data structure manipulations, desugaring language constructs, generating type constraints and optimizing ASTs. Summary of our benchmarks can be found in Figure 6. We also show the differences between using input-output examples versus generalized harnesses wherever possible. In addition, we explore how small modifications to the sketch can affect the scalability. All our benchmarks can be found in the supplementary material.

Bench	Description	cbits	Distinct choices
runEx	Running Example	75	2^{14}
runEx_c	Running example as in Example 3.2	160	2^{15}
treeEx	Insertion in a binary tree using examples	1830	2^{72}
treeGen	Insertion in a binary tree using general harness	1830	2^{72}
lang	Simple language desugaring	913	2^{110}
langState	Simple language extended with state	501	2^{141}
lcB	Booleans to Lambda Calculus	3505	2^{541}
lcB_e	Booleans to Lambda Calculus with extra information	390	2^{136}
lcP	Pairs to Lambda Calculus	842	2^{183}
lcP_e	Pairs to Lambda Calculus with expanded desugar	414	2^{161}
tcEx	Type constraints for lambda calculus with examples	1058	2^{149}
tcUni	Type constraints for lambda calculus with unification algorithm	877	2^{144}
astOpt	AST Optimizations	1160	2^{162}

Figure 6. Summary of benchmarks

Figure 6 also shows the number of bits in the control vector of each sketch (cbits) and number of distinct choices of the sketch (computed manually). Because of don′t-cares and symmetries, the number of distinct choices is smaller than 2^{cbits}, but it is still very large. The complexity of the synthesis problem depends on these distinct choices, the amount of coupling between different cases in pattern matching and the harnesses. For problems like the running example, it is possible to reason about every case independently, but this is not the case for all our benchmarks.

6.1.1 Insertion in immutable binary tree

This benchmark synthesizes insertion into an immutable binary tree. The type definitions and the sketch of insertNode method are shown in Figure 7. This sketch first recurses on its children and then uses a GUC to generate the new BinaryTree after insertion. The sketch has more recursive calls than necessary, but these will not impact correctness because of immutability. The minrepeat construct indicates that a minimal set of if-statements should be synthesized.

This benchmark can either be constrained by a few concrete input-output examples (as in treeEx) or by asserting correctness for general trees (as in treeGen). The benchmark treeEx requires 5 input-output examples of trees of depth 2 (leaves correspond to depth 1) to fully constraint the sketch. The benchmark treeGen uses a produce function to non-deterministically generate trees of maximum depth 2, and checker functions to check that the tree will be correct after inserting an additional non-deterministic value. It can be seen from Figure 10 that the second approach is almost 21 times slower than the first.

6.1.2 Desugaring language constructs

We explored four different kinds of benchmarks that involve synthesizing desugaring from a high level language to a low level language with two of them involving translations to lambda calculus. All these benchmarks are constrained by general harnesses that assert equivalence of interpreted source and destination languages. Therefore, these benchmarks require implementation of interpreters for source and destination languages and an equals function that asserts equivalence between their outputs. In addition, they also require a produce function to generate symbolic source language ASTs.

```
adt BinaryTree {
  Branch { int value; BinaryTree l; BinaryTree r; }
  Leaf {int value;}
  Empty {}
}

BinaryTree insertNode(BinaryTree tree, int x) {
  if (tree == null) return null ;
  switch(tree){
    repeat_case:{
      minrepeat{
        if ({| x (<|>|<=|>=) tree.?? |} ) {
          BinaryTree l = insertNode(tree.??, x);
          BinaryTree r = insertNode(tree.??, x);
          return ??(3, {tree.??, l, r, x});
} } } } }
```

Figure 7. BinaryTree ADT representation and sketch of insertNode method

```
dstAST desugar(srcAST s){
  switch(s){
    repeat_case: {
      dstAST a = desugar(s.??);
      dstAST b = desugar(s.??);
      dstAST c = desugar(s.??);
      return ??(3, {a, b, c, s.??});
} } }
```

Figure 8. Sketch of desugar function for lang benchmark

Simple language desugaring This benchmark is a simple extension to the languages used in running example. Here, both source and destination language contain Unary Prim1S/D, Binary Prim2S/D and If IfS/D statements. The differences are source language has separate TrueS and FalseS nodes whereas destination contains a single BoolD node. Also, the source language contains an extra construct BetweenS {srcAST a; srcAST b; srcAST c;}which evaluates to TrueS if $a < b < c$, otherwise FalseS. The sketch of the desugar function for this benchmark is shown in Figure 8.

In this benchmark, our harness checks srcASTs of depth 2 and it is sufficient to fully constraint the sketch. One possible implementation obtained for desugaring of BetweenS is shown in pseudocode below.

```
BetweenS (a= a, b= b, c=c) -> Prim2D( op = new Oand(),
    a = new Prim2D( op = new Olt() , a = desugar(s.a) ,
            b= desugar(s.b)),
    b = new Prim2D( op = new Olt() , a = desugar(s.b),
            b = desugar(s.c)))
```

Simple language extended with state This benchmark extends the previous one by adding LetS/D and mutable state that can be modified using AssignS/D. The desugaring of BetweenS obtained before is incorrect when b modifies the state since it is desugared twice. A correct desugaring must use LetD to avoid this problem.

Synthesizing this translation is trickier than the previous case for two reasons: (1) A single node BetweenS is translated to a dstAST with at least depth 5 and the solution space increases exponentially with depth. (2) There is heavy coupling between the variants because of the mutable state.

In order to synthesize the code for this benchmark in reasonable amount of time, we increase of bounds on generated ADTs only for the case of BetweenS. And also with some knowledge about how the desugaring should look like, we also expand the GUC to Unknown Constructor for the BetweenS case and use different depths for different recursive children.

Desugaring booleans to lambda calculus It is also interesting to look at how we can synthesize Church encodings for some constructs. The benchmark lcB synthesizes translation for booleans and operations on booleans. The source language contains the variants TrueS, FalseS, AndS, OrS and NotS and the destination language is lambda calculus with variants for variables, abstraction and application.

This benchmark is more complex than the previous ones due to the complexity of the required interpreters. In these benchmarks, we used the call-by-name interpreter. In order to compare the outputs of the interpreters for srcAST and dstAST, the harness needs to know the encodings for TrueS and FalseS. This also allows the solver to generate strong constraints on desugaring of TrueS and FalseS early on and solve for AndS, OrS, and NotS cases separately. The desugar function, here, is sketched similar to lang benchmark shown earlier, but it instead constructs dstASTs till depth 6. The version lcB_e is similar to lcB except that in lcB_e, we give extra information to solver by giving the encodings for TrueS and FalseS in the GUC and cutting down depth required to 3. Both these harnesses check srcASTs to depth 2.

Desugaring pairs to lambda calculus The benchmark lcP synthesizes translation from a language supporting pairs (PairS) and operations on pairs (FirstS and SecondS) to simple lambda calculus. Here, apart from the complexity of lambda calculus interpreter, we also do not assume the encoding for PairS. Therefore, in this case, the desugaring of FirstS and SecondS depends on the desugaring of PairS and vice-versa. Also, the correctness guarantees of encodings of these constructs is achieved together rather than independently as in the previous benchmarks. This benchmark uses dstASTs of depth 4 in the desugar function (similar to lang benchmark) and the harness checks for srcASTs of depth 3.

The version lcP_e expands the GUC used in the desugar function into an Unknown Constructor and constructs the children using GUCs of depth 3. We have seen in example 3.2 that this change will reduce the number of choices.

6.1.3 Type constraints for lambda calculus

This benchmark synthesizes an algorithm to produce type constraints for a lambda calculus AST to be used in order to do type inference. The output of the sketch is a conjunction of type equality constraints which the algorithm produces by traversing the AST. There are two versions of this benchmark. First, using input-output examples to constrain the sketch as in tcEx. This approach has a disadvantage that constructing the input-output pairs is very tricky and not as simple as constructing input-outputs in case of insertion into a binary tree. Therefore, the second version (tcUni) uses the unification algorithm to simplify the type constraints and only asserts equivalence between the expected final type and inferred type after unification. The second version still uses input-output examples but the outputs here are greatly simplified. Both these benchmarks have 8 harnesses to constrain the sketch.

6.1.4 Synthesizing optimizations on ASTs

In this benchmark, we use SYNTREC to produce optimizations on ASTs that are used internally to represent the sketch in SYNTREC. Here, we explore ASTs constructed out of Numbers, Booleans, operators on them and Multiplexer. Given a set of possible ASTs that can be optimized, this benchmark finds a predicate for each one them and if the predicate is satisfied, it generates an optimized formula and verifies that both the optimized version and original AST generate same outputs. We use GUCs to generate both the predicates and the optimized ASTs. In this benchmark, we harness optimizations on 4 different ASTs.

6.2 Experiments

Methodology All our experiments were run on a 10-core Intel Xeon E5-2470 @ 2.40GHz machine. We ran each experiment ten times and report the median.

Hypothesis 1: Synthesis of complex routines is possible It is clear that the above benchmarks are very complex in terms of the search space. However, we show that we can synthesize them in reasonable amount of time in SYNTREC. The results on running these benchmarks can be seen in the first 2 columns of Figure 10.

Hypothesis 2: Recursive-Tuple-based encoding is better than Object encoding The last 2 columns of Figure 10 shows the results of running the benchmarks using the object-encoding (section 4.3) of algebraic datatypes. We can clearly see that both runtime and memory consumption are significantly smaller for recursive tuple based encoding compared to object encoding for most of the benchmarks.

Hypothesis 3: Comparison with standard SMT encoding We also conducted an experiment to test our encoding of tuples with the Z3 SMT solver using recursive datatypes and records to represent tuples. For each benchmark, we captured the constraints generated for the inductive synthesis problem

```
dstAST desugar(srcAST s, int bnd){
  if (s == null || bnd < 0){ return null; }
  dstAST a = desugar(s.a, bnd−1);
  dstAST b = desugar(s.b, bnd−1);
  minrepeat{
    if (s.type == ??){
      return new dstAST(type = ?? , op = ??, a = {|a|b|},
      b = {|a|b|}, val = {|s.val | ?? |}, v = ?? );
} } }
```

Figure 9. Desugar function for running example using struct

at every step of CEGIS and used a script to translate from our internal format into Z3's input format. At this point, the problem has already been inlined and simplified by our solver. Figure 11 shows for each benchmark the aggregate solution times for all inductive synthesis problems for both solvers. This translation uses Z3 theory of integers, arrays, and datatypes. We also extracted the output from Z3 results for the benchmarks that don't timeout and verified it on the sketch.

We noticed that Z3 is comparable to SYNTREC in verification problems even though SYNTREC uses only 5-bit inputs and Z3 uses theory of integer linear arithmetic.

Hypothesis 4: Type information from Algebraic Datatypes significantly reduces the search space Another way to write all our benchmarks is to use a struct to define all variants of the ADT (with a field *type* to distinguish between different variants) and use if-else statements on *type* instead of pattern matching. Figure 9 shows the desugar function for running example in this version . Here, minrepeat{} block minimizes the number of if-blocks of the form if (s.type = ??) and {|...|} represents regular expressions. This sketch is at a comparable level of abstraction as the one that uses repeat_case, but it is much more difficult to resolve. Also note that, most of the constructs described in Section 3 are not applicable in this case and the sketch must hard code these constructs. Especially, it may take a great deal of time to write an efficient hard-coded form for GUC.

We compared the run times of our benchmarks with this struct representation and the results can be seen in Figure 12. It is clear that this approach does not scale on most of our benchmarks. The reason is that whereas repeat_case can take advantage of type information to specialize each case for a given variant, in the sketch above the synthesizer has to discover what the cases are and how many there should be.

If we enumerate the above if (s.type == ??) to all cases manually rather than using the minrepeat, the runtimes are close to our current system, but this approach makes the sketch very verbose.

Benchmark	Recursive data types		ADT as objects	
	Runtime	Memory	Runtime	Memory
runEx	3.25	91.59	2.88	91.59
runEx_c	3.69	91.59	2.49	110.44
treeEx	7.44	204.32	788.61	4585.12
treeGen	157.85	1381.55	1791.14	7012.32
lang	60.06	882.26	535.16	2676.83
langState	648.39	1962.79	TO	TO
lcB	517.92	1747.54	TO	TO
lcB_e	5.89	248.21	1251.66	7077.06
lcP	1662.12	1919.72	TO	TO
lcP_e	945.88	1656.61	TO	TO
tcEx	10.23	261.88	17.55	590.07
tcUni	496.12	1743.62	TO	TO
astOpt	163.09	880.44	108.44	556.06

Figure 10. Run times(s) and Memory consumption(MiB) for various benchmarks (TO > 2700s)

Bench	SYNTREC	Z3
runEx	0.005	0.06
runEx_c	0.048	3.19
treeEx	3.39	287.97
treeGen	145.14	TO
lang	47.01	TO
langState	577.19	TO
lcB	514.21	2323.43
lcB_e	4.07	370.28
lcP	1508.80	4141.05
lcP_e	938.92	1394.26
tcEx	0.95	215.78
tcUni	489.24	TO
dagOpt	137.52	TO

Figure 11. Synthesis times(s) comparison with Z3. TO>max(700, 2* SYNTREC time)/iteration

Bench	ADT	struct
runEx	3.25	0.59
runEx_c	3.69	0.63
treeEx	6.32	17.49
treeGen	40.35	2152.49
lang	20.07	206.83
langState	755.20	TO
lcB	368.02	TO
lcB_e	5.89	1428.44
lcP	945.88	TO
lcP_e	123.03	TO
tcEx	10.87	340.63
tcUni	496.12	TO
dagOpt	163.09	84.38

Figure 12. Run time(s) comparison for ADT vs normal structs (TO>2700s)

6.3 Scalability

Currently, our system is scalable to routines that generate ADTs of depth 3 or 4. Therefore, we need to have stricter bounds on the GUCs. However, we should also note that our system fails quickly if insufficient bounds are set. Hence, a quick bottom-up approach can be followed to reach the optimal bounds.

Another problem to scalability arises due to function inlining. Most functions in the benchmarks (including the helper functions) are highly recursive and inlining all functions to a certain depth (default 5) blows up the size of the formula used to represent the programs. In this respect, the optimization mentioned in section 4.2 helps to some extent. But, we also need to have stricter inlining limits if possible. In SYNTREC, it is possible to set a inlining limit for the entire sketch which is used in some of our benchmarks. Here also, a sketch with insufficient inlining limits will fail quickly, and in some cases, the output error also indicates that it failed because the inlining was not sufficient.

In cases where generated ADTs are deeper and the amount of coupling between different variants is also higher, the user may have to give extra information to help scalability. For example, if we know that a particular AST appears in its full form in a larger AST, we can pass it in the GUC. This is illustrated in the benchmark lcB_e. Figure 10 show that this benchmark performs significantly better than its counterpart lcB. Also, expanding the GUC into a Unknown Constructor for one level (as done in the benchmark lcP_e) will scale better. In SYNTREC, it is also possible to explicitly have special cases inside **switch** and have the rest figured out by repeat_case construct. This is useful when there are only a few variants that have non-trivial transformation and requires producing deeper ASTs. This trick is used in langState benchmark.

In addition, having harnesses with deeper symbolic ASTs does not scale very well because the verification phase need to verify on a large number of ASTs. In these cases, it is better to have a mix of concrete harnesses of deeper ASTs and symbolic ASTs of smaller depths.

7. Related Work

The most relevant piece of related work is the synthesizer Leon by the LARA group at EFPL [Blanc et al. 2013; Kneuss et al. 2013; Kuncak 2014], which builds on prior work on complete functional synthesis by the same group [Kuncak et al. 2010]. In particular, their recent work on Synthesis Modulo Recursive Functions [Kneuss et al. 2013] 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.

There has been a lot of recent work on programming by example systems, some of it focusing explicitly on recursive programs. For example, there is recent work by Albarghouthi, Gulwani and Kincaid in using an explicit search technique to synthesize recursive programs from examples [Albarghouthi et al. 2013]. 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 [Perelman et al. 2014]. 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. Both of these projects, however, are limited to programming-by-example settings, and cannot deal with the kind of test harnesses that we use in some of our experiments.

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 [Solar-Lezama 2008]. However, several other solver based synthesizers have been reported in the literature, such as Brahma [Gulwani et al. 2011]. The work on proof theoretic synthesis [Srivastava et al. 2010] 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 [Srivastava et al. 2011], 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 [Torlak and Bodík 2014, 2013] 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 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 [Suter et al. 2010]; 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, Spima and Manna developed a decision procedure to solve combinations of Presburger Arithmetic and term algebras [Zhang et al. 2006], and showed how to use this procedure to model balanced trees [Manna et al. 2007]. Their work, in turn, built on the work of Oppen on decision procedures for recursively defined data-structures [Oppen 1980]. Today, several of the most popular SMT solvers support reasoning about recursive data-structures [De Moura and Bjørner 2008; Barrett et al. 2007; Dutertre and De Moura 2006]. 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.

8. Conclusion

The paper has shown that by combining type information from algebraic data types with 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. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. In B. Cook and R. Sebastiani, editors, *Combined Proceedings of the 4^{th} Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '06) and the 1^{st} International Workshop on Probabilistic Automata and Logics (PaUL '06)*, volume 174(8) of *Electronic Notes in Theoretical Computer Science*, pages 23–37. Elsevier, June 2007. Seattle, Washington.

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. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489838. URL http://doi.acm.org/10.1145/2489837.2489838.

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 195–204, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287653. URL http://doi.acm.org/10.1145/1287624.1287653.

B. Dutertre and L. De Moura. The yices smt solver. 2006.

N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

V. Ganesh, C. W. O'Donnell, M. Soos, S. Devadas, M. C. Rinard, and A. Solar-Lezama. Lynx: A programmatic sat solver for the rna-folding problem. In *SAT*, pages 143–156, 2012.

S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000. doi: 10.1145/347324.383378. URL http://doi.acm.org/10.1145/347324.383378.

E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426, 2013.

V. Kuncak. Verifying and synthesizing software with recursive functions - (invited contribution). In *ICALP (1)*, pages 11–25, 2014.

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.

Z. Manna, H. B. Sipma, and T. Zhang. Verifying balanced trees. In *Proceedings of the International Symposium on Logical Foundations of Computer Science*, LFCS '07, pages 363–378, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72732-3. doi: 10.1007/978-3-540-72734-7_26. URL http://dx.doi.org/10.1007/978-3-540-72734-7_26.

D. C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, July 1980. ISSN 0004-5411. doi: 10.1145/322203.322204. URL http://doi.acm.org/10.1145/322203.322204.

D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, page 43, 2014.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100.

A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

A. Solar-Lezama. Open source sketch synthesizer. 2012. URL https://bitbucket.org/gatoatigrado/sketch-frontend/.

A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS '06*, San Jose, CA, USA, 2006. ACM Press.

S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. *POPL*, 2010.

S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, pages 492–503, 2011.

P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. *SIGPLAN Not.*, 45(1):199–210, Jan. 2010. ISSN 0362-1340. doi: 10.1145/1707801.1706325. URL http://doi.acm.org/10.1145/1707801.1706325.

E. Torlak and R. Bodík. Growing solver-aided languages with rosette. In *Onward!*, pages 135–152, 2013.

E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54, 2014.

M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 505–520, 2003. doi: 10.1007/3-540-36577-X_37. URL http://dx.doi.org/10.1007/3-540-36577-X_37.

T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. *Information and Computation*, 204(10):1526 – 1574, 2006. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2006.03.004. URL http://www.sciencedirect.com/science/article/pii/S0890540106000630. Combining Logical Systems.