PLDI: U: Type Assisted Synthesis of Recursive Transformers on Algebraic Datatypes

Jeevana Priya Inala
MIT
jinala@mit.edu
Advisor: Armando Solar-Lezama

1. Problem and Motivation

As programming languages are being developed to be used in a wide range of industrial applications, it is necessary to introduce convenience language constructs that abstract a lot of the underlying low-level code. For example, the `for` statement in languages like Java and C is a convenience construct that can be purely expressed in terms of the `while` statement. Compilers for these languages, however, have a set of core language constructs which makes it easier for them to do further analysis and optimizations. Therefore, the first step that a compiler does is to transform these convenience constructs to the core constructs–a process known as "desugaring". Any typical compiler has numerous such desugaring transformations and these are often very large and difficult to write correctly. For instance, one of the research compilers that is being developed in my group has hundreds of desugaring passes which on total account for about 10,000 lines of code. Hence, in this work, we want to explore if automated program synthesis can help generate these transformations directly from the semantics of the languages and thus, reducing a lot of burden off the programmers. The goal of program synthesis is to simplify programming by allowing users to express their intent as specifications over non-deterministic inputs and the synthesis tools search for a candidate program (in this case, the correct transformation) that satisfies these specifications.

In fact, these kinds of desugaring transformations fall into a very general category of programs. We can treat a program as an expression tree with each node representing a language construct–this representation is known as Abstract Syntax Tree (AST). Desugaring is the transformation from one AST to another AST that semantically preserves the meaning of the original program. Moreover, AST is a special type of the more general category called Algebraic Datatypes (ADT). For example, a binary tree ADT is shown below:

```
adt BTree {
  Branch { int v; BTree l; BTree r; }
  Leaf { int v; }
  Empty {} }
```

Our goal is to develop a program synthesis tool for this general class of programs i.e. synthesizing transformations on recursive algebraic datatypes. Other examples of ADT transformations include end user programming like data structure manipulations, figuring out formula simplifications for expression ASTs (used in compiler optimizers) and generating constraints for type inference (used in program analysis tools). A natural way of specifying these programs is to establish a functional relation between the input ADT and the expected transformed ADT. For example, the correctness specification for desugaring languages can be defined in terms of the output on an interpreter on the original and desugared languages.

The main problem in synthesizing recursive ADT transformations is that the search space is extremely huge. Hence, it is necessary to have an efficient way of describing the space of possible implementations that the synthesizer should consider and an efficient search strategy. Recently, Syntax-guided-synthesis (SyGus) [1] has gained popularity as an approach to synthesis as it leverages a syntactic description of the space of possible programs (called as template) to improve the scalability of synthesis. The motivation behind SyGus is to allow programmers to specify the high level structure of the program they are interested in while leaving out the low-level error prone regions of the code for the synthesis tool to figure out. The SyGus approach works well for our problem if the search space encoded by the template is as compact as possible. However, writing these efficient templates is hard and hence, there is not much gained benefit if users have to write a template for every synthesis problem.

Our key insight is that it is possible to make these templates highly reusable for the domain of ADTs by relying on the type information from ADTs. The re-usability 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. We also describe a new optimization called *inductive decomposition* to aid the synthesis of recursive transformations on ADTs. Inductive decomposition leverages the specification inductively to break the synthesis problem into smaller and easy to solve synthesis problems. The efficiency gained by this optimization, together with the ability of our system to benefit from user guidance, allows our system to attack problems that could not be synthesized by other related systems. We show, for example, that our system can automatically infer desugaring functions for simple languages including lambda calculus representation for pairs and booleans just from the language interpreters. 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.

2. Background and Related Work

Sketch: SKETCH [6] is a state of the art tool for generating programs from specifications which are given as partial C like programs. Sketch specifications contain special "holes" denoted by "??" that represent arbitrary integers (or booleans). The purpose of SKETCH tool is to instantiate these "holes" with fixed values so that all assertions and constraints specified in the program are satisfied. SKETCH does this by using a constraint based approach (CEGIS) and uses a SAT solver internally.

Our work extends the SKETCH language by introducing high level holes (synthesis constructs) that operate on algebraic datatypes and using bi-directional type checking to transform these high level holes to just integer holes.

Leon: The most relevant piece of related work is the synthesizer Leon [4] which can synthesize provably correct recursive functions involving algebraic datatypes. Even though, it is possible to describe the functional specifications like interpreters in Leon, it does not scale well on these kinds of benchmarks. One reason for this is that Leon lacks support for user defined space of programs and hence, has to search for programs from a very general space of programs. On the other hand, this allows Leon to be more automatic. Our technique of using re-usable spaces of programs achieves the best of both worlds.

Other synthesis tools: There has been a some recent work on type directed synthesis of programs on ADTs [2, 5], but, these are limited to programming-by-example settings, and cannot deal with the kind of complex specifications (for example, involving interpreters) that we use in our benchmarks. Rosette [9] is a solver aided language that has shown how to embed synthesis capabilities in DSLs. However, Rosette is a dynamic language and lacks static type information, so its templates are not re-usable.

3. Uniqueness of Approach

We introduce SYNTREC which is implemented as an extension to the open source SKETCH synthesis system [7]. We make the following contributions:

```
adt srcAST{
  NumS{ int v; }
  TrueS{ }
  FalseS{ }
  BinaryS{ opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; } }
adt dstAST{
  NumD{ int v; }
  BoolD{ bit v; }
  BinaryD{ opcode op; dstAST a; dstAST b; } }
adt opcode{ AndOp{} OrOp{} LtOp{} GtOp{} ... }
```

Figure 1: ADT for two small expression languages

- We define new type-directed synthesis constructs (TDC) which when combined with *polymorphic generators* allow programmers to express the high-level structure of the intended program in a re-usable manner while enabling the synthesizer to derive the low-level details.
- We use bi-directional type checking rules to efficiently reduce the new synthesis constructs.
- Finally, we develop a novel optimization called inductive decomposition that further enhances the scalability of the system.

3.1 Running example

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. Here, the trickiest desugaring is that of the $BetweenS$ variant which computes $a < b < c$ and needs to represented in terms of $BinaryD$ in the destination AST.

The first step to synthesize a function is to specify its intended behavior. In the case of desugaring functions, the best way to specify their behavior is by establishing the equivalence of their respective interpreters. Specifically, our language allows us to state a constraint of the form

```
assert ( interpretS (exp) === interpretD(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 interpretS and interpretD are two interpreters written in SYNTREC and defined recursively over the structure of the respective ADTs. As we will explain in Section 3.3, 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.

The programmer must also describe a space of possible implementations from which the desugar function must be synthesized. This space is described through a *template*, a partial program that describes the high-level structure of the solution while leaving the details unspecified. Thanks to the new constructs presented in this paper, 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 templates.

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.

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

In the code above, recursiveReplacer is a *polymorphic generator* [6] that is defined in a library and it represents a space of programs parameterized by the type of src. The code for recursiveReplacer is shown below.

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

A generator is like a hygienic macro, and can be thought of as a function that gets inlined and specialized to its calling context at compile time. The generator above describes a very general computational pattern. It contains several of the new synthesis constructs that are described in the more detail in Figure 3. This generator represents a recursive replacer (with holes for the low-level details that need to be figured out by the synthesizer) that pattern matches over the input AST src and in each case, recursively calls the function rec on its children and finally generates a new AST expression of type Q whose terminals will either be constants or the results of the recursive calls or some fields of src. Type Q is determined by the calling context of this generator. For the running example, our system takes about 50s to synthesize this code and a compacted version of the 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.

3.2 Type Directed Synthesis Constructs

For the domain of algebraic datatypes, three operations are very important–pattern matching, accessing fields and constructing new ADTs. Our type directed constructs (as shown in Figure 3) exactly capture these operations on ADTs. The running example illustrated some of these type-directed constructs (TDCs). Each of these constructs represents a set of

```
dstAST desugar(srcAST src) {
  switch(src) {
    case NumS: return NumD(v = src.v);
/* Some cases are abstracted to save space */
    case BetweenS:
      dstAST[3] arr = { desugar(src.a), desugar(src.b),
              desugar(src.c) };
      return  BinaryD(op = AndOp(),
     a = BinaryD(op = LtOp(), a = arr[0], b = arr [1])
     b = BinaryD(op = LtOp(), a = arr[1], b = arr [2]));
}}
```

Figure 2: Synthesized solution for the running example

$repeat\_case$	:=	Expands into a pattern matching for each variant
$s.??$	:=	Returns an arbitrary field of s
$s.\{T\}$	:=	Returns an array of all fields of s of type T
$??(e_1, ..., e_n)$	:=	Constructs an ADT expression using the arguments as terminals

Figure 3: Type directed constructs

possible expressions in the language, but the key is that the exact set is determined by the type of expressions passed to the TDC, and by the type expected by the context in which the TDC is used. The running example also showed that these type-directed constructs can be particularly useful when applied in the context of *polymorphic generators*, because then the generator can be used with many different types, and the space of program fragments that the generator can produce will depend on the types of its arguments. These TDCs can also be combined with *polymorphic generators* to create richer constructs such as, for example, iterators over data structures.

In order to define the semantics of these new synthesis constructs, we define a type directed transformation $\Gamma \vdash e \xrightarrow{\theta} e'$ which reduces all the new constructs to sets of expressions in the base language. Here, Γ is the environment that tracks types of variables in the program. The goal is to have the synthesizer choose among this set of expressions for one that satisfies the specification. This is done by using unknown integer/boolean constants, since the semantics of programs with unknown constants have been well described in previous work [8]. The type θ that parameterizes the transformation is used to propagate information top-down about the type of the expression that is required in a given context and thus, making the rules bi-directional. Detailed transformation rules for all these rules can be found in the tech report [3]. The following example shows how bi-directional type inference and expansion work for a field selector hole ($s.??$).

EXAMPLE 3.1. *Consider the program below*

```
adt A {int l1; int l2; A l3; A l4;}
A x;  int y = x.??;  A z = x.??
```

Since the types of y and z are int and A respectively, we need to evaluate the following

$$\Gamma \vdash x.?? \xrightarrow{int} E_1 \qquad \Gamma \vdash x.?? \xrightarrow{A} E_2$$

Here, the bottom-up type information flows through x which is of type A and top-down type information flows from the expected context (i.e. the type on the judgment arrow). Here, the ADT A has two fields of type int, namely $l1,\ l2$, and has two fields of type A, namely $l3,\ l4$ and hence, the expansion produces the following results where the choice *construct requires the synthesizer to choose among these field access expressions.*

$$\Gamma \vdash x.?? \xrightarrow{int} choice\{l1, l2\} \quad \Gamma \vdash x.?? \xrightarrow{A} choice\{l3, l4\}$$

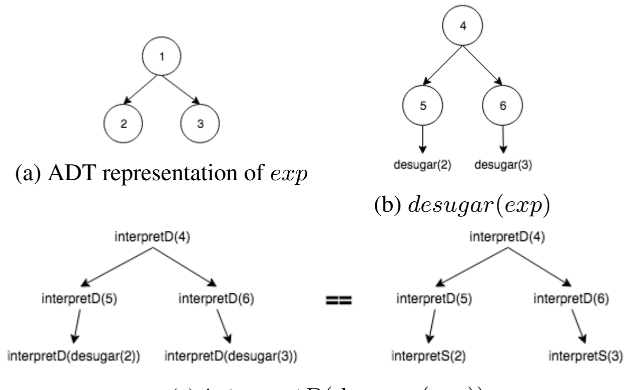
3.3 Synthesis

The bi-directional type inference and expansion rules from Section 3.2 allow us to reduce the synthesis problem to a problem of synthesizing integer/boolean unknowns. Solutions to this problem have been described for simple imperative programs, but solving for these choices in the context of algebraic datatypes and highly recursive programs poses some new challenges that have not been addressed by prior work.

3.3.1 Inductive Decomposition

Our system follows the standard CEGIS approach to solve for the integer unknowns that work for any non-deterministic inputs (bounded) [6]. For readers unfamiliar with this approach, the most relevant aspect from the point of view of this work is that the problem is reduced to a sequence of inductive synthesis steps where the system tries to generate solutions that work for small sets of inputs, followed by checking steps that try to determine whether a correct solution has been found. One of the problems with directly using CEGIS for highly recursive problems is that the synthesizer will have to reason about the entire function to be synthesized during every inductive step in CEGIS. Consider the running example. The $desugar$ code for each variant of the $srcAST$ is an unknown piece of code which the synthesizer is trying to discover. At each iteration of the CEGIS algorithm, there is a concrete value exp of type $srcAST$ and the synthesizer is adding constraints to enforce that the specification is satisfied for that concrete value exp. Now under the given specification, a given concrete exp will exercise multiple variants within $desugar$. This means that the synthesizer has to jointly search for all of these variants.

The main idea of inductive decomposition is to inductively use the specification to allow the synthesizer to reason about the code for each variant separately and thus, greatly improving the scalability of CEGIS. This idea is depicted in Figure 4. Normal CEGIS approach would have directly expanded the recursive $desugar$ calls in Figure 4b. Instead, we delay the expansion until it is required by the $interpretD$ calls (Figure 4c). In most cases, this expansion is not required because we can use the specification to inductively replace $interpretD(desugar(s))$ with $interpretS(s)$ and thus, decoupling the variants from one another.



(a) ADT representation of exp

(b) $desugar(exp)$

(c) $interpretD(desugar(exp))$

Figure 4: Illustration of inductive decomposition on the running example.

4. Results and Contributions

We evaluated our approach on 22 benchmarks as shown in Figure 5. Most of these benchmarks are beyond the scope of what can be synthesized by other tools like Leon, and others. It can be seen from Figure 5 that SYNTREC can synthesize these functions in a couple of seconds to a couple of minutes. All of these benchmarks are synthesized from templates containing 2-5 lines of code using a library with only two different generators. Some of these benchmarks are discussed in more detail below.

4.1 Desugaring language constructs

Desugaring to lambda calculus We all know that lambda calculus is a Turing-complete language and can be used to express high level constructs like booleans, integers, pairs and lists. However, the details of these translations are very tricky. Indeed, these are usually given as assignments in introductory program analysis courses and we found it is very difficult for students to get the translations right. However, our tool can synthesize the translations for booleans (lcB) and pairs (lcP) down to pure lambda calculus in less than 2 minutes.

***Desugaring passes in* SKETCH** Apart from some toy languages, we used SYNTREC to synthesize some simple desugaring passes in SKETCH. For example, the benchmark compAssign desugars compound assignments such as $x+ = 1$ into $x = x+1$. A more interesting benchmark is arrAssertions which represents a compiler pass in SKETCH that adds array out of bounds assertion checks for every array access.

4.2 Lists and trees manipulations

We have also used SYNTREC to synthesize several data structure manipulations such as inserting or deleting an element from lists and trees. Figure 6 shows the synthesized

	Bench	Description	Run time
Desugar	bet	Running example	51.3
	betState	Running example with mutable state	556.5
	regex	Desugaring regular expressions	3.3
	elimBool	Boolean operations to if else	2.0
	compAssign	Eliminates compound assignments	58.7
	mergeOp	Merge operations into one construct	85.6
	arrAssertions	Add out of bounds assertions	177.9
	lcB	Boolean operations to lambda calculus	24.2
	lcP	Pairs to lambda calculus	109.4
Analysis	tc	Type constraints for lambda calculus	175.9
Formula Simp	andLt	Formula simplification 1	2.2
	andNot	Formula simplification 2	1.6
	andOr	Formula simplification 3	2.9
	plusEq	Formula simplification 4	7.8
	mux	Formula simplification 5	1.7
List	lIns	List insertion	8.3
	lDel	List deletion	10.5
	lUnion	Union of two lists	8.3
Tree	tIns	Binary search tree insertion	487.5
	tDel	Binary search tree deletion	269.0
	tDelMin	Binary search tree delete min	129.5
	tDelMax	Binary search tree delete max	132.0

Figure 5: Benchmarks. All reported times are in seconds.

```
BTree insert(BTree tree, int x) {
  switch(tree){
    case Branch:
        BTree l = insert(tree.l, x);
        BTree r = insert(tree.r, x);
        if (x <= tree.v)
          return Branch(v = tree.val, l = l, r= tree.r)
        else
          return Branch(v = tree.val, l = tree.l, r = r)
    case Leaf:
      if (x <= tree.v)
          return Branch(val = tree.v, Leaf(v = x), null)
        else
          return Branch(val = tree.v, null, Leaf(v = x))
    case Empty:
      return Leaf(val = x)     }}
```

Figure 6: Solution for insertion into a binary search tree

solution for insertion into an immutable binary search tree from a 2 line template that uses the same recursiveReplacer library function as in Section 3.1.

4.3 Type constraints for lambda calculus

This benchmark synthesizes an algorithm to produce type constraints for lambda calculus language 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 which can then be solved to discover the types of all expressions in the AST.

4.4 Formula Simplifications

In these benchmarks, we use SYNTREC to produce simplification rules on AST formulas that are used in the internal representation of programs 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 simplified, this benchmark finds a predicate for each one them and if the predicate is satisfied, it generates an simplified formula and verifies that both the optimized version and original AST generate same outputs. Here is an example rule generated by our system:

$$\text{equal(plus(a,b), plus(c,d))} \xrightarrow{equal(a,c)} \text{equal(b,d)}$$

5. Conclusion

The paper has shown that by combining type information from algebraic datatypes together with the inductive decomposition optimization, it is possible to efficiently synthesize complex functions based on ADTs, including desugaring functions for high level constructs down to lambda calculus. Even though, synthesizing full desugaring passes or other compilation transformations for larger languages is beyond the scope of the current work, we believe that the work in this paper is an important step in that direction.

References

[1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[2] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[3] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015.

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

[5] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

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

[7] A. Solar-Lezama. Open source sketch synthesizer. 2012.

[8] 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.

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