Abstract
We argue that synthesizing operations on recursive linked data structures is not as hard as it appears and is, in fact, within reach of current SAT-based synthesis techniques—with the addition of a simple approach that we describe to decompose the problem into smaller parts. To generate smaller pieces of code, i.e., shorter routines, is obviously easier than large and complex routines, and, also, there is more potential for automating the code synthesis.

In this paper, we present a code generation algorithm for synthesizing operations of linked data structures and, as an example, describe how the proposed algorithm works to synthesize operations of an AVL tree.

Categories and Subject Descriptors D.1.2 [Programming Techniques]: Automatic Programming; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Languages

Keywords code generation algorithm, program synthesis, SAT-solver, linked data structures, AVL trees

1. Introduction
Software synthesis has received renewed attention in recent years. Advances in SAT/SMT solvers and verification technology have been directed at synthesis challenges. Great progress has been made; nevertheless, synthesizing operations on linked data structures appears just out of grasp as such operations usually require either recursion or loops.

We demonstrate that a careful decomposition of the problem can bring it within reach of current SAT-based synthesis techniques (e.g., [7, 8, 10]). The key insights are:

- operations on linked data structures typically navigate only a small number of links;
- operations on linked data structures can be decomposed into a number of simple cases based on the values observable by a limited number of dereferences;
- each of these simple cases requires a limited number of straight-line statements (i.e., no conditionals and no loops), the last of which might be a recursive call;
- a small number of necessary helper methods, such as pointer swaps, can be easily synthesized (and automatically specified) without regard to the specific operations to be performed on the overall structure;
- some synthesized methods might produce results that violate some invariants, which are then repaired by a different synthesized method;
- the choices of which invariants to violate (and repair) can be made automatically by grouping the data structure’s invariants according to the fields that they refer to.

We demonstrate this approach with a case study of the AVL tree [1]. An AVL tree is a self-balancing tree, and as such it requires some tricky rebalancing methods [2]. It is common practice for programmers to write insert and delete operations for self-balancing trees that preserve the ordering of the tree but violate the balancing invariants; a rebalancing operation is then called to repair the balancing invariants (while preserving the ordering invariants). Our synthesis technique also follows this practice, and is able to do so automatically by grouping the invariants into those that preserve balancing and those that preserve order.

We work with programs written in Java and specifications written in JFSL [3, 13], which is a variant of the Alloy relational logic [6] suitable for specifying functional correctness properties of sequential Java programs.

The paper is organized as follows. §2 describes our approach and the observations and insights it is based on. §3 demonstrates our approach on the AVL tree example. §4 provides a discussion on applicability of our approach to other data structures. Related work is discussed in §5, and §6 concludes.

2. Approach
In this section, we describe the common pattern that is the key to our code generation approach and present our overall algorithm. The key insights that enable this pattern are that operations on linked data structures typically navigate only one link at a time, and that therefore an operation on a linked data structure can be broken down into a finite number of cases based on the values of the immediately observable fields.

2.1 Common Pattern
One of the central findings of our work is that there exists a code pattern that can be seen in the code for all the linked data structures’ operations. It is presented in Figure 1. All the operations can be represented as a sequence of if-blocks containing a number of...
The if-blocks are branched on a conjunction of several conditions that are mathematical comparisons, that is, expressions comparing the right-hand side to the left-hand side using the mathematical signs, such as "$\leq\$, "$\geq\$, "$\lt\$, "$\gt\$, and "$\neq\$. In some cases, the condition statements also might use the $+$ and $-$ arithmetic operations. Each statement inside the blocks of straight-line code is one of the following:

- a recursive call to the method itself,
- a call to a different method (a helper method or another method to be generated),
- an assignment of a new value to some existing variable,
- a variable declaration, or
- an assignment of a value to the introduced variable.

In the pattern, for each method, the number of conditions in each if-statement is the same for each block, but some of the conditions might be redundant and can be eliminated on refactoring. Also, the synthesized code could be automatically refactored to merge conditionals that lead to identical straight-line blocks, and such a refactoring would make the code nicer for a programmer to read.

### 2.2 Code Generation Algorithm

As a starting point for code generation, we need the programmer to specify the basics. The programmer should:

1. Define the basic data structures (e.g., `Node`).
2. Write invariants for the linked data structure as a whole.
3. List method names for the methods to be synthesized and pre- and post-conditions for each of them.

After this information is provided, the following steps are taken to generate the linked data structure's operations (all of these actions can be automated):

1. Generate helper methods (and their specifications) that are based solely on the definitions of the data structures and not on the specifications of the methods to be synthesized.

For example, when dealing with AVL trees, this step would generate a method to swap the left and right children of a node.

2. Generate helper methods (and their specifications) based on the data structure invariants.

3. Generate the conditional statements (see template in Figure 1) by taking the cross product of the possible variables and values.

4. For each conditional statement, use a SAT solver to generate the straight-line block of statements for it.

5. Clean up by consolidating blocks that have the same body.

The SAT solver in step 4 is searching for a sequence of statements that satisfy the method post-conditions given that the conditions for the block, the method pre-conditions, and the invariants are true. The search space contains five types of statements as was described above (subsection 2.1). These SAT solver runs are independent and may be run in parallel.

The main focus of the invariant analysis is dividing the provided invariants into groups to determine potential properties of the linked data structure that should be accounted for separately in the code of the methods to be generated. Essentially, the number of groups of invariants can correspond to the number of helper methods inside the main methods. The division of the invariants may be based on different properties, for example, on the attributes mentioned in each invariant as it is in the forthcoming example.

Code generation for different types of methods takes different values as fillers. Thus, for generating a `@Pure` method, i.e., a method that does not modify the overall linked data structure, the template is filled only with values from the method’s pre- and post-conditions. `@Modifies` methods may be broken into several helper methods according to the result of the invariant analysis. Then, the main method to be generated will contain that number of helper methods, even if it is just one. When generating a helper method for a main `@Modifies` method, the filler values for the code generation are taken from one specific group of invariants and the pre- and post-conditions for the main method. Once the helper methods are generated, the main method is essentially a correctly working permutation of the helper methods.

### 3. Synthesis of AVL Tree Operations

In this section, we show in detail how our proposed code generating algorithm works to generate operations of an AVL tree in Java.

An AVL tree is a self-balancing binary search tree with the property that, for each node, the difference in height between its left child and its right child is at most one [1].

The programmer must supply the definitions of the data structures (Figure 2), the invariants, and the pre- and post-conditions for the methods to be synthesized. (Due to space limitations, the invariants and pre- and post-conditions are not shown.)

#### 3.1 Generating Code

Once the essential information about the linked data structure is supplied, automatic code generation procedure can begin. Our current prototype is a very limited proof of concept: we construct an Alloy model and read out the results manually. All helper methods are generated in a systematic manual procedure. This prototype is not usable, but it illustrates that the essential ideas work.

**Swap’ Methods** Code generation starts with creating the simplest methods—‘swap’ methods. These methods are generated based only on the definition of the `Node` structure and basically take the `Node`’s fields of appropriate types and swap their values. For example, `swapChildren` method, shown on Figure 3, takes a `Node` as a parameter and assigns its left child to be the right child and its right child to be the left child. Similar methods are gen-

```java
public return-type method-name(arguments) {
    if (condition1 && condition2 && ... && conditionN) {
        statement1;
        statement2;
        ...
        statementM;
        return object-of-the-return-type-or-nothing;
    }
    ...
    if (condition1 && condition2 && ... && conditionN) {
        statement1;
        statement2;
        ...
        statementM;
        return object-of-the-return-type-or-nothing;
    }
}
```

### Figure 1. Code Generating Pattern

```java
class Node {
    int value, height;
    Node left, right, parent;
}
```

### Figure 2. Definition of Node Data Structure
erated for the other fields of the Node structure. Calls to ‘swap’ methods are used as possible executable statements in if-blocks of the template for all the methods to be generated.

Specifications for these methods are also generated and then used by subsequent synthesis steps.

We can now synthesize the contains method according to the pattern in Figure 1. In total the contains method has 24 conditional blocks, each of which has one statement. A further refactoring could merge conditions whose blocks contain the same statement so that the code would be more human readable.

Invariants Analysis Before proceeding to generate more complicated non-pure methods (marked modifies in JFLS), i.e., the methods that change the AVL tree, such as insert and delete methods, we need to determine the number of properties the data structure has. We do that by performing an invariants analysis.

The invariants analysis is done on the invariants as specified by the programmer and based on the fields names of the Node structure (Figure 2) also as defined by the programmer.

The invariants can be grouped according to which fields they mention. The left and right fields are used ubiquitously, but the value and height fields are used more selectively: value is used by invariants that control tree ordering, whereas height is used by invariants that control tree balancing. We treat these groups of invariants separately, as do programmers in practice.

Insertion Operation Based on the result of the invariant analysis, the non-pure insert method consists of two helper methods: one concerned with the order of nodes, which we call insertHelper(), and one concerned with maintaining the tree in balance, which we call balanceHelper(). First the helper methods are generated according to the template, and then the order in which to call them is determined. The main insert method is just two statements: calls to these helper methods.

The insertHelper method contains 48 conditional blocks, whereas the balanceHelper has 768.

Deletion Operation The synthesis of the delete operation is analogous to that for the insertion operation, with an interesting variation: the generated code for the insert method does not use other AVL tree operations, whereas the generated code for the delete method does—it uses an insertion operation.

The most tricky case to handle when deleting a node from a binary search tree is when a node to be deleted has two children. The standard coding practice uses additional helper methods to find either the node’s in-order successor, i.e., the left-most child of the node’s right subtree, or in-order predecessor, i.e., the right-most child of the node’s left subtree [2]. However, our synthesis technique produced an alternative using an insert operation instead.

A code snippet simplified in the number of conditions inside the if-statement is shown below.

```java
if (node.left != null && node.right != null && node.value == x) {
    node.value = node.right.value;
    Node temp = node.right.left;
    node.right = null;
    } else { // other cases...

Figure 3. swapChildren Method
```

3.2 Analysis

The key insight that facilitates our divide and conquer approach is that operations on linked data structures typically look at just one link at a time and at most at two, so the set of possible conditions that they can examine is finite and not that large. Within each of these conditional blocks only straight-line code is needed (possibly with a recursive call), and that can be synthesized with known SAT-based techniques (e.g., [7, 8, 10]). By analyzing the code synthesized in this AVL case study, we can get a sense of how large the search space is. We first consider the generation of the conditionals:

- In accordance with the algorithm description, conditions statements inside the if-blocks use the mathematical comparison operators, such as ==, !>, <, and >. On the left- and right-hand sides, Node fields, null, and numeric values from the specification are used.
- The conjunction inside the if-statements usually has 3 to 5 conditions, and, only in one case (the balanceHelper method), it has 9 conditions. The number of conditions is determined by the variables in scope and the number of fields they have.

In this AVL case study there are, at most, on the order of $2^{10}$ conditional blocks generated.

The synthesis of the straight-line code within each conditional block is independent, and explores a space of the following size:

- The kinds of executable statements include:
  - a recursive call to the method itself,
  - a call to a different method (e.g., ‘swap’ methods inside the balanceHelper method and the insert method inside the deleteHelper method),
  - a declaration of a variable,
  - an assignment to the declared variable, and
  - an assignment of a new value to a Node field.

- The straight-line code blocks usually have 1 to 6 statements, and, only in one case (the balanceHelper method), it has 12 statements.
- Up to three levels of dereferencing (i.e., node.child1.child2.f) is necessary in any kind of statement. However, there is only one method where three levels of dereferencing is needed—the balanceHelper method; the rest of the methods need only two levels of dereferencing (i.e., node.child1.f).

The search scope for these straight-line blocks can start small and be expanded if a solution is not found (e.g., [11]). For example, starting with one statement and one level of dereferencing gives a space on the order of $2^6$. Expanding to a second level of dereferencing increases the space to the order of $2^{18}$, depending on how many fields are in the relevant classes (e.g., Node). Increasing to six statements gives on the order of $2^{18}$ and $2^{30}$ for one and two levels of dereferencing respectively. These spaces are within reach of modern SAT solvers.

On the other hand, the balanceHelper method has blocks that require up to twelve statements and three levels of dereferencing. This method might push the limits of what is possible with current techniques.

```java
void swapChildren(Node one) {
    Node temp = one.right;
    one.right = one.left;
    one.left = temp;
}
```
The search can be guided and reduced by the order in which different types of statements are introduced. Our observation is that the following order would be effective:

1. A recursive call to the method itself.
2. An assignment of a new value to a field.
3. A call to a different generated method.
4. A declaration of a variable and an assignment to it.
5. A call to a ‘swap’ method.

In summary, while the number of conditional blocks might be in the hundreds, they are generated simply and easily, and provide the scaffolding for decomposing the problem into smaller cases that can be solved independently. The straight-line code within each block is usually just a few statements, and in most cases requires only one level of field dereferencing. The most complex straight-line blocks occur within `balanceHelper`, which require twelve statements and three levels of field dereferencing.

4. Applicability to Other Data Structures

We selected AVL trees as a case study of a reasonably complex linked data structure, and we expect that our approach generalizes to similar kinds of structures, such as unbalanced binary search trees and linked lists. The described code generation algorithm is suitable for recursive linked structures with known data layouts that navigate one link at a time.

Our approach, in its current form, is not suitable for array-based structures, such as array-lists or hash tables. The potentially large number of indices and the possibility of doing arithmetic on the indices both present challenges to the proposed algorithm.

Tree-like structures that make use of arrays and index computations, e.g., B-trees, van Emde Boas trees, and Fibonacci heaps, are also beyond the current capabilities of our approach.

Likewise, handling structures, such as tries (prefix trees), where the data to be stored are decomposed according to some computation exceeds what the algorithm described here can do.

Skip lists, while recursive linked structures, are outside the reach of our approach because the different levels of links require more complex conditionals and loops in the code.

5. Related Work

Gulwani [4] surveys the literature on program synthesis and sees three dimensions to it: how the user’s intent is expressed, the search space, and the search technique. According to these dimensions, our work classifies as follows. From the five different ways the user can express their intent, in our approach, the user expresses their intent primarily by writing specifications in the JFSL/Alloy relational logic language [3, 6, 13]. We also require the user to write the basic data layout of the classes. From the four different search spaces identified by Gulwani, our approach generates imperative programs in a stylized form: conditionals are generated according to a template; there are no loops; straight-line blocks are generated with known SAT-based techniques and may include recursive calls. Finally, from the four kinds of search techniques, our approach uses a combination of exhaustive search and logical reasoning. We exhaustively generate all possible conditionals from a finite set of combinations and, within each conditional, use logical reasoning to synthesize the appropriate block of straight-line code.

To Gulwani’s classification [4] we add a fourth dimension: the data manipulated by the program. Gulwani’s survey considers techniques that synthesize programs that manipulate integers, bit vectors, or sets. There was relatively little work done on synthesizing programs that manipulate heap references at the time of his survey, and this is the category that our approach falls under.

In our work, user intent is expressed in logical specifications, and we search in the space of imperative programs using logical reasoning techniques. This kind of approach has received an increasing amount of attention in recent years, e.g. [5, 7, 8, 12]. Even so, relatively little of this work deals with programs that manipulate heap references.

Leino and Milicevic [8] proposed a dynamic synthesis code generation algorithm that adopts ideas from concolic testing and combines concrete and symbolic execution, which is rather different from our approach. In addition, their algorithm works towards shrinking the possibly unbounded search space on each iteration while our algorithm starts with a smaller search space and expands it if the solution is not found on the current iteration. Furthermore, their currently presented work, although allows recursion, is limited to generating just read-only methods whereas our algorithm can generate methods modifying the underlying data structure.

Kuncak et al. [7] present a synthesis procedure for Boolean Algebra with Presburger Arithmetic (BAPA), which is used to manipulate sets and their sizes. Their approach significantly differs from ours, firstly, because it relies on integer linear arithmetic, and, secondly, because it does not go beyond sets and it is unclear whether it can be applied to other data structures. Moreover, the code generated by their synthesis procedure uses built-in libraries for sets and operations on them, which we restrain ourselves from.

In previous work [9], we took a syntactic approach to generating iterators over complex heap structures; however, these iterators do not modify the heap.

6. Conclusion

We have presented a divide-and-conquer strategy to the synthesis of operations on linked data structures. A case study on AVL trees illustrates that the problem might not be as difficult as had been previously imagined. The key insights are that operations on linked data structures typically examine a small number of links in a limited way and that within each conditional block only straight-line code is needed. These observations lead to search spaces that should be manageable by current SAT-based synthesis techniques.

References