Natural Synthesis of Provably-Correct Data-Structure Manipulations

Abstract

This paper presents natural synthesis, which generalizes the proof-theoretic synthesis technique to support very expressive logic theories. This approach leverages the natural proof methodology and reduces an intractable, unbounded-size synthesis problem to a tractable, bounded-size synthesis problem, which is amenable to be handled by modern inductive synthesis engines. The synthesized program admits a natural proof and is a provably-correct solution to the original synthesis problem.

We explore the natural synthesis approach in the domain of imperative data-structure manipulations and present a novel syntax-guided synthesizer based on natural synthesis. The input to our system is a program template together with a rich functional specification that the synthesized program must meet. Our system automatically produces a program implementation along with necessary proof artifacts, namely loop invariants and ranking functions, and guarantees the total correctness with a natural proof. Experiments show that our natural synthesizer can efficiently produce provably-correct implementations for sorted lists and binary search trees. To our knowledge, this is the first system that can automatically synthesize these programs and their total correctness in tandem from bare-bones control flow skeletons.

1. Introduction

Building programs with formal correctness guarantee is highly desirable in today's software development, and automating this process is one of the central themes in programming languages research. Ideally, the goal is, from a very rich specification of a programming task written in pre/-post annotations, to automatically produce a verified program satisfying the specification. Such a tool can avoid the traditional "program-and-verify" paradigm and alleviate the programmer's burden of debugging back and forth between the program and its proof artifacts (loop invariants, ranking functions, etc.). Despite of being a long time dream since the late 1970s [15], this goal typically can only be achieved for finite programs [21, 24] or with user interaction [4, 10].

In recent years, due to the significant progress in automated constraint solving, *proof-theoretic synthesis* [22] was proposed to automate the synthesis process. While encoding the synthesis task as a set of constraints with unknowns, this approach relies on a verification oracle to solve the synthesis conditions, which limits the applicability of this paradigm to decidable theories such as arrays and arithmetics. For more sophisticated logic theories, the underlying verifier may be unavailable or not efficient enough.

For example, due to the inherent complexity of automated reasoning about heap structures and data, no existing systems can automatically synthesize imperative data-structure manipulations and guarantee their functional correctness. On the one hand, the state-of-the-art verification systems [2, 8, 9, 13] can verify many sophisticated data-structures; but none of them is synthesis-enabled, i.e., the user needs to provide a full program along with modular contracts, loop invariants and ranking functions. On the other hand, automated synthesis of provably-correct data-structure manipulations has received less attention; the state-of-the-art systems either produce functional programs [4, 10] or support only weak specifications in the form of input/output samples [20].

In this paper, we present *natural synthesis*, a novel approach to automated program synthesis, which aims to generalize proof-theoretic synthesis to handle more sophisticated logic theories. The insight behind this approach is borrowed from *natural proofs* [5, 14, 17, 18]. As a *sound but incomplete* verification technique, the essence of natural proofs is to identify and automate a fixed set of simple but useful proof tactics, and algorithmically search for a proof that only uses these tactics, which called a natural proof. Due to the simplicity and usefulness of these tactics, the search process is decidable, very efficient and tends to be successful.

Similar to natural proofs, the goal of natural synthesis we present in this paper is to enable automated synthesis of provably-correct programs from rich and complex specifications. To follow this approach, the user should pick or define a powerful logic, with respect to which the synthesis problem is immediately intractable. Then the user should identify a set of natural proof strategies and aim to find a program that admits a natural proof. By doing so the original synthesis problem is *soundly reduced* to a *natural synthesis problem* with a stronger specification, i.e., any program synthesized from the new specification also meets the original specification. Moreover, the natural synthesis problem should fall into simpler logic theories, which can be easily encoded and handled by today's inductive synthesis engines such as SKETCH [21] or ROSETTE [24], resulting in a fully automatic synthesis algorithm.

This paper focuses on synthesizing imperative provably-correct data-structure manipulations over dynamically-allocated heaps. Following the natural synthesis methodology, we develop a syntax-guided program synthesizer that can produce imperative, provably-correct data-structure manip-

ulations. The input to the synthesizer is a program template written in IMPSYNT , a heap-manipulating, synthesis-enabled language. IMPSYNT allows the user to describe only a high-level skeleton of the program, and leave implementation details as unknown holes. The user may also specify pre-/post-conditions for each function using formulas in DRYAD^{FO} logic, an expressive first-order logic extended with user-defined recursive definitions. The output is a complete, verified program along with all auxiliary annotations used for verifying the program's total correctness, including loop invariants and ranking functions. The salient features of the program synthesizer set forth in this paper include:

a) supports rich functional specifications written in a very expressive program logic;

b) synthesizes desired implementations (e.g., recursive or iterative) from user's high-level guidance;

c) guarantees full functional total correctness by synthesizing implementations and proof artifacts in tandem;

d) produces automatically and efficiently verified programs manipulating both standard list/tree data-structures.

To our knowledge, this is the first program synthesizer that can achieve all the above goals simultaneously.

In the rest of this paper, we present the technical details of our synthesizer and demonstrate how the the natural synthesis approach is applied. Section 2 shows the synthesis-enabled language IMPSYNT and the specification logic DRYAD^{FO}, and how programmers can easily and precisely describe their programming tasks. Section 3 explains how natural proofs can help to build simple proofs, and summarizes the specific tactics for recursive data-structures. Section 4 presents the main idea of natural synthesis: reduce the intractable problem of synthesizing unbounded manipulations on an unbounded-size heap to a tractable problem of synthesizing *bounded* manipulations on a *bounded-size, symbolic* heap. Section 5 formulates the natural synthesis problem using logic theories including arithmetic, arrays, uninterpreted function, etc., which are commonly supported by modern program synthesizers. Section 6 reports experiments on synthesizing common imperative manipulations on data-structures such as sorted lists and binary search trees. Most notable algorithms that can be automatically synthesized and verified include insertion sort, BST insertion, root removal, etc. Given such sophisticated imperative data-structure manipulations automatically synthesized and verified, with minimal user inputs and such rich functional correctness guaranteed, we believe natural synthesis can take the automaticity of synthesizing data-structure manipulations to a new level.

2. Overview

In this section, we first introduce the language IMP and its extension, IMPSYNT , which allows describing holes and specifications. Then we explain how our natural synthesizer works through a concrete example.

2.1 A Language for Data-Structure Manipulation

In this paper, we focus on synthesizing basic data-structure manipulations, and define a simple language that allows structured control flow built up from statements of five categories: assignments, mutation, allocation, deallocation and return. Basically, IMP is a type-safe language that is similar to a subset of C in both syntax and semantics; its semantics mimics C semantics on normal executions, but is more strict than C on runtime errors. In other words, the produced IMP programs can be easily converted to C programs that still guarantee the proved correctness.[1]

IMP also allows the programmer to specify preconditions, postconditions, loop invariants and ranking functions in DRYAD^{FO}, a variant of the DRYAD logic [14], excluding sets/multisets. Intuitively DRYAD^{FO} specifies both structural and data properties in a first-order logic extended with recursive definitions. We will present more details of DRYAD^{FO} in Section 3.1.

The grammar of IMP is presented as in Figure 1. To simplify the presentation, we assume that there is only one struct type defined in a program, with a set of pointer fields *PF* and a set of data fields *DF*.

Our system allows the programmer to provide a program sketch, as well as a set of logic specifications. The system then automatically synthesizes an implementation of the sketch which is verified to satisfy the given specifications. Both the program sketch and the specification are written in a language called IMPSYNT , which extends IMP with *unknown variables*, *fields*, *conditions*, *statements*, and even *loops* and *function calls*. A hole can be of one of the following four forms:

1. The special symbol `??` represents an arbitrary unknown constant, variable or field, and can be placed at any context where a variable or field is required;
2. `cond(`C`)`/ `val(`C`)` represents an unknown condition/expression; the constant C indicates the maximum size of the syntax tree.
3. `stmt(`C`)` represents an arbitrary, loop-free, unknown code snippet consists of up to C statements, in which the branch conditions, statement type and the involved variables are all unknown.
4. `conj(`C`)`/ `exp(`C`)` represents a conjunction/int term in DRYAD^{FO}; again, C is the maximum size of the unknown formula/expression.

IMPSYNT also allows the user to describe even higher level unknown code snippets. For example, to create a new node in the heap and initialize each field of the node with unknown values, the programmer can simply write

[1] Except the out-of-memory error; see the semantics defined in Section 4.1.

$$
\begin{array}{rrl}
\text{Program } p & ::= & \epsilon \mid p\,;\,p \mid T\ f(arg)\ \{\texttt{requires}\ \varphi\,;\ \texttt{ensures}\ \varphi\,;\ \texttt{decreases}\ it\,;\ blk\} \\
\text{Arguments } arg & ::= & \epsilon \mid arg\ ,\ arg \mid \texttt{loc}\ u \mid \texttt{int}\ j \\
\text{Block } blk & ::= & \epsilon \mid blk\,;\ blk \mid \texttt{if}\ (cnd)\ \{blk\}\ \texttt{else}\ \{blk\} \mid \texttt{while}\ (cnd)\ \{\texttt{invariant}\ \varphi\,;\texttt{decreases}\ it\,;blk\} \\
\text{Cond. } cnd & ::= & exp == var \mid exp < var \mid \texttt{!}\,cnd \mid cnd\ \texttt{\&\&}\ cnd \\
\text{Stmt. } st & ::= & T\ var := exp \mid var.fld := var \mid \texttt{loc}\ u := \texttt{malloc()} \mid \texttt{free}(u) \mid \texttt{return}\ var \\
 & & \mid T\ var := f(var_1, \ldots, var_n) \\
\text{Type } T & ::= & \texttt{loc} \mid \texttt{int} \\
\text{Expr. } exp & ::= & var \mid var.fld \mid \texttt{nil} \mid C \mid exp + exp \mid exp - exp \mid cond\ ?\ exp : exp \\
\text{Var. } var & ::= & u \mid \ldots \mid j \mid \ldots \\
\text{Field } fld & ::= & dir \mid \ldots \mid df \mid \ldots
\end{array}
$$

f : Function Name u : Loc Var. j : Int Var. $dir \in PF$ $df \in DF$ C : Int Const
it : Int. Term in DRYAD$_{\text{simp}}$ φ : DRYAD$_{\text{simp}}$ Formula (allowing the `old` construct)

Figure 1: Grammar of IMP

```
loc v := new;
```

Assuming the struct type contains k fields, then this statement is just syntactic sugar for

```
loc v := malloc();
v.fld_1 := ??;
...
v.fld_k := ??;
```

Similarly, the programmer may also describe an unknown while-loop (including initializing statements before the loop) as a hole of the form

```
loop(V, N)
```

where V and N estimate the number of mutable `loc` and `int` variables involved in the loop, respectively. More concretely, `loop(V, N)` is syntactic sugar for the following very general template:

```
if (cond(1)) loc lv_1 := val(2);
...
if (cond(1)) loc lv_V := val(2);
if (cond(1)) int iv_1 := val(2);
...
if (cond(1)) int iv_N := val(2);
while (cond(1)) {
  invariant preserves_old() && conj(V + 1);
  decreases exp(1);
  if (cond(1)) lv_1 := val(2);
  ...
  if (cond(1)) lv_V := val(2);
  if (cond(1)) iv_1 := val(2);
  ...
  if (cond(1)) iv_N := val(2);
  if (cond(1)) lv_1.?? := ??;
  ...
  if (cond(1)) lv_V.?? := ??;
}
```

where `preserves_old()` is a conjunction of condition for preserving old values: for each term $\texttt{old}(f^*(t))$ appeared in the final postcondition, we assume the old value preserved as an term $\texttt{exp}(2)$, i.e., the loop invariant contains a formula $\texttt{old}(f^*(t)) = \texttt{exp}(2)$.

IMPSYNT also allows `simple-loop(V, N)` as a simpler alternative, which does not allow the destructive updates to `lv_1` through `lv_V` at the end of the loop body. Note that the constants V and N only indicate the number of variables updated within the loop; other variables can still be accessed in the loop as long as they are not updated. Therefore these constants are usually small numbers such as 2 or 3. The programmer may estimate a reasonable number, or start from a small number and increment it if the synthesizer fails to find a solution.

To write a program template, IMPSYNT requires all `requires` and `ensures` formulas are fully specified; as pre-/post-conditions of functions, they should always be provided by the user. IMPSYNT also expects the user to convey her high level design decisions, in terms of a high-level control flow graph and/or the site of each loop/function call. For a single programming task, the programmer may pick different approaches to implement it, e.g., iterative or recursive, and IMPSYNT can discover appropriate implementation details such that the synthesized program can be verified. Now let us use a running example to explain the usage model, how a programmer can describe her programming tasks in IMPSYNT , and what our system can automatically produce.

2.2 An Example: Insertion to A Sorted List

Consider the process of developing a program to insert a new key `k` into a sorted list with head `h`. The programmer first needs to logically specify the expected behavior of the program using preconditions and postconditions. In this example, at the beginning of the program, she may claim that `h` should be the root of a sorted list, and the output should still be a sorted list. Moreover, the output list should have one more key than the old list; the max (min) key is either the old max (min) key or `k`, whichever is greater (less). She can write the above specification logically as a `requires` and an `ensures` in the program body, as shown in Figure 2a. `MAX` and `MIN` are shorthands for corresponding `ite`-terms in DRYAD$^{\text{FO}}$. The formulas involve several recursively-defined predicates and functions: $sorted\_l^*$, len^*, min^* and max^*; we will explain how to formally define them shortly in the next section. While in this example, these common recursive definitions are pre-defined in our system, in general, the pro-

```
loc srtl_insert_iter(loc h, int k) {
  requires sorted_l*(h);
  ensures sorted_l*(ret) ∧ len*(ret) = old(len*(h)) + 1
  ∧ max*(ret) = MAX(old(k), old(max*(h)))
  ∧ min*(ret) = MIN(old(k), old(min*(h)));

  if (cond(2)) { //trivial case
    loc ?? := new; //create a new node and return
    return ??;
  }
  else {
    //find the right position
    simple-loop(2, 0);
    loc ?? := new; //create a new node
    stmt(1); //insert
    return ??; //return
  }
}
```

(a) Input: An iterative program template in IMPSYNT

```
loc srtl_insert_iter(loc h, int k) {
  requires sorted_l*(h);
  ensures sorted_l*(ret) ∧ len*(ret) = old(len*(h)) + 1
  ∧ max*(ret) = MAX(old(k), old(max*(h)))
  ∧ min*(ret) = MIN(old(k), old(min*(h)));

  if (h == nil || h.key >= k) {
    loc n := malloc();
    n.key := k;
    n.next := nil;
    return n;
  }
  else {
    loc v_1 := h;
    loc v_2 := h.next;
    while(v_2 != nil && v_2.key < k) {
      invariant sorted_l*(h) ∧ len*(h) = old(len*(h))
        ∧ min*(h) = old(min*(h)) ∧ max*(h) = old(max*(h))
        ∧ k = old(k) ∧ sorted_lseg*(h, v_2)
        ∧ v_1.next = v_2 ∧ v_1.key ≤ k;
      decreases len*(v_2);
      v_1 := v_2;
      v_2 := v_2.next;
    }
    loc v_3 := malloc();
    v_3.key := k;
    v_3.next := v_2;
    v_1.next := v_3;
    return h;
  }
}
```

(b) Output: A verified complete program in IMP

Figure 2: Synthesizing a sorted-list-insertion program (iterative implementation)

grammer can define arbitrary recursive predicate or function for her custom data structures and programming tasks.

Now to allow the system to synthesize an implementation that meets the above specification, the programmer may start with a rough idea: in the trivial case, i.e., the list is empty, simply create a new node with the new key and return it; otherwise, the program needs to find the appropriate position to

```
loc srtl_insert_rec(loc h, int k) {
  requires sorted_l*(h);
  ensures sorted_l*(ret) ∧ len*(ret) = old(len*(h)) + 1
  ∧ max*(ret) = MAX(old(k), old(max*(h)))
  ∧ min*(ret) = MIN(old(k), old(min*(h)));

  if (cond(2)) { //trivial case
    loc ?? := new; //create a new node and return
    return ??;
  }
  else {
    stmt(1); //find the sublist
    //recursively insert into the sublist
    ?? := srtl_insert_rec(??, ??);
    stmt(1); //connect to the new sublist
    return ??;
  }
}
```

(a) Input: A recursive program template in IMPSYNT

```
loc srtl_insert_rec(loc h, int k) {
  requires sorted_l*(h);
  ensures sorted_l*(ret) ∧ len*(ret) = old(len*(h)) + 1
  ∧ max*(ret) = MAX(old(k), old(max*(h)))
  ∧ min*(ret) = MIN(old(k), old(min*(h)));
  decreases len*(h);

  if (h == nil || h.key >= k) {
    loc n := malloc();
    n.key := k;
    n.next := nil;
    return n;
  }
  else {
    loc v_1 := h.next;
    v_1 := srtl_insert_rec(v_1, k);
    h.next := v_1;
    return h;
  }
}
```

(b) Output: A verified complete program in IMP

Figure 3: Synthesizing a sorted-list-insertion program (recursive implementation)

which the newly created node can be inserted. The programmer may decide to implement the non-trivial case as a while-loop or a recursive function call. We discuss both situations as follows.

Iterative implementation. To find the position of insertion iteratively, the programmer may simply write a while-loop template `simple-loop(2,0)`, with both the loop condition and the loop body unknown. Note that this loop is simply for searching a position and does not update any variable, hence `simple-loop` is sufficient, and the number of integer variables involved is 0. After finding the right position, the programmer may create a node-creating template `loc n := new` followed by an unknown statement `stmt(1)` that is expected to finish the insertion, as shown in Figure 2a. Finally, an unknown variable `??` will be returned. The trivial-

case branch is relatively simpler: create a new node and return it.

Now given as input the program template as in Figure 2a, our system automatically fills all the holes, namely, the branch condition, the statement, the loop condition and loop body, the return variable. More importantly, our system also finds a proof of the synthesized program's *total correctness*, by generating a loop invariant and a ranking function. In other words, all these artifacts are synthesized appropriately such that for *arbitrary* input `h` and `k` satisfying the precondition, the loop invariant will be established and maintained, and finally the execution will terminate and the postcondition will hold. In addition, our system also guarantees that runtime errors such as null dereference are absent from the synthesized program. Figure 2b shows the complete program synthesized by our system. The loop condition is an inequality check between a location's key field and an integer variable, and the body consists of two variable-update statements. As the most impressive part, the synthesized loop invariant consists of 8 conjuncts and combines recursive predicates, arithmetic and old values from function entry, etc., in a very sophisticated way. The found ranking function is the length of the list starting from `v_2`, which always decreases and guarantees the termination of the loop.

Recursive implementation. Alternatively, the programmer may also construct a verified recursive implementation from a template. Figure 3a shows a recursive template of the sorted-list-insertion program. The trivial case is similar to its counterpart in the iterative version. However, the non-trivial case becomes entirely different: the centerpiece is a recursive function call to `srtl_insert_rec` itself, whose arguments are unknown, and store the location to an unknown variable. In addition, an extra statement is needed before the function call to find the sublist for function call; and another statement is needed after the function call to integrate the returned new sublist to form a new sorted list.

From this template our system can synthesize a recursive implementation, as shown in Figure 3b. Similarly, our system not only fills all the holes in the template, but also proves the total correctness of this program w.r.t. the function's contract. Note that a ranking function is claimed as in the new **`decreases`** clause of the function's contract, and also proved.

3. Natural Proofs for DRYAD$^{\mathbf{FO}}$

In the previous section, we have given an overview of our program synthesizer and how it can help a programmer build a verified sorted-list insertion program. As mentioned in Section 1, the synthesis technique behind our system is called natural synthesis. As a general synthesis methodology, the first step of natural synthesis is to define an expressive logic as the specification language, and identify a set of natural proof tactics. In this section, we briefly introduce the natural proof methodology, and present necessary formalism of DRYAD$^{\text{FO}}$ and its natural proof tactics, and intuitively, how natural proofs work through our running example.

Natural Proofs [5, 14, 17, 18] is a proof methodology proposed to ease the inherent tension between the logic expressiveness and the reasoning automaticity. Given an expressive logic, the basic strategy of natural proofs is to develop automatic, terminating but incomplete reasoning procedures by:

a) From the class of all possible proofs identify a subclass of proofs that mimic human's manual proof tactics, called natural proofs;

b) Search the above class of proofs thoroughly by terminating procedures, typically powered by automatic and efficient logic solvers.

The first natural-proof-based technique were developed by [14] for DRYAD$_{tree}$ to verify tree data-structure manipulations. [18] extended natural proofs for a separation logic DRYAD$_{sep}$ to handle more general properties of structure, data, and separation. The natural proof technique has also been encoded into ghost code and integrated with the VCC verifier [3]. In this paper, our specification logic DRYAD$^{\text{FO}}$ is a variant of previous DRYAD logics excluding sets/multisets.

3.1 The DRYAD$^{\mathbf{FO}}$ Logic

DRYAD$^{\text{FO}}$ is designed to specify heap structure and data properties using two sorts: *Loc* for locations in the heap and *Int* for integers. The syntax of DRYAD$^{\text{FO}}$ is shown in Figure 4. It is quantifier-free but allows recursive definitions of various types such as integers, sets, multisets, etc. It allows recursive predicates of the form $p^*(x)$ or $p^*(x, y)$, or recursive functions of the form $i^*(x)$ or $i^*(x, y)$, where p^* and i^* are recursively defined on the first argument x.

As shown in Figure 4, a unary recursive function is defined using the syntax $i^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil},\ i_{base},\ i_{ind})$, where i_{base} and i_{ind} are themselves terms that stand for what i^* evaluates to when $x = \texttt{nil}$ (the base-case) and when $x \neq \texttt{nil}$ (the inductive step), respectively. For example, recall that our running example `srtl_insert` involves four recursive definitions: *sorted_l**, *len**, *min** and *max**. The length of a list can be recursively defined in DRYAD$^{\text{FO}}$:

$$\mathit{len}^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil}, 0, \mathit{len}^*(x.\texttt{next}) + 1)$$

And the definition of *sorted_l** relies on *min**:

$$\mathit{sorted\_l}^*(x) \stackrel{def}{=} \texttt{ite}\big(x = \texttt{nil},\ \texttt{true},\ \mathit{sorted\_l}^*(x.\texttt{next}) \wedge x.\texttt{key} \leq \mathit{min}^*(x.\texttt{next})\big)$$

Other predicates and functions can also be easily defined.

DRYAD$^{\text{FO}}$ also allows binary recursive predicates/functions to characterize partial data structures such as list segment from x to y. For example, *sorted_lseg*$^*(x, y)$ defines a sorted list segment, and *len_lseg*$^*(x, y)$ defines the length of the list segment. In these definitions, the base case condition

$dir \in PF$ $\quad i^* : Loc \to Int$ $\quad x, y \in Loc$ Variables $\quad c : Int$ Constant

$f \in DF$ $\quad p^* : Loc \to \{\texttt{true}, \texttt{false}\}$ $\quad j \in Int$ Variables $\quad q \in Boolean$ Variables

$$\textit{Loc Term: } lt, lt_1, lt_2 \ldots ::= x \mid \texttt{nil} \mid lt.dir$$

$$\textit{Int Term: } it, it_1, it_2, \ldots ::= c \mid j \mid lt.f \mid i^*(lt) \mid i^*(lt, y) \mid it_1 + it_2 \mid it_1 - it_2 \mid \texttt{ite}(\varphi, it_1, it_2)$$

$$\text{Formula: } \varphi, \varphi_1, \varphi_2, \ldots ::= \texttt{true} \mid q \mid p^*(lt) \mid p^*(lt, y) \mid lt_1 = lt_2 \mid it_1 \leq it_2 \mid \texttt{disjoint}(x, y) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2$$

$$\text{Recursively-defined function : } i^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil},\ i_{base},\ i_{ind}) \quad \text{or} \quad i^*(x, y) \stackrel{def}{=} \texttt{ite}(x = y,\ i_{base},\ i_{ind})$$

$$\text{Recursively-defined predicate : } p^*(x) \stackrel{def}{=} \texttt{ite}(x = \texttt{nil},\ p_{base},\ p_{ind}) \quad \text{or} \quad p^*(x, y) \stackrel{def}{=} \texttt{ite}(x = y,\ p_{base},\ p_{ind})$$

Figure 4: Syntax of DRYAD$^{\text{FO}}$ Logic

becomes $x = y$. For technical reasons, there is a set of syntactic restrictions on these definitions, which can be found in Appendix A.

The semantics of DRYAD$^{\text{FO}}$ is defined on a graph model of the heap. Intuitively, we model the heap as a finite graph: each record in the heap is represented as a node; and each pointer field is represented as an edge labelled with the corresponding field name. The semantics is formally defined in Appendix A. The most noteworthy part is the semantics of recursive definitions. To guarantee the recursive definitions can be eventually reduced to the base case, DRYAD$^{\text{FO}}$ requires the portion of the heap used in evaluating a recursive definition to form a tree. For example, in the `srtl_insert` example, the precondition $sorted\_l^*(\texttt{h})$ implicitly requires `h` to be the head of a non-circular list; otherwise the the formula is not valid.

3.2 Proof Tactics for DRYAD$^{\text{FO}}$

When DRYAD logics were developed in [14, 18] for automated program verification, two proof tactics inspired by human's manual proofs were identified: a) unfold recursive definitions across the footprint of the program (the locations explicitly dereferenced in the program); and b) to make the recursive definitions uninterpreted. The two tactics were shown to be practically useful in verifying a large variety of data-structure algorithms. To formally describe these tactics, we extend the formalism proposed in [14], called *symbolic heap*. A symbolic heap can be viewed as an abstraction, representing arbitrarily large concrete heaps using bounded-size graphs, and is amenable to conducting symbolic execution; we leave the formal definitions in Appendix B. the main difference from [14] is that their formalism does not support loops. Moreover, we extend the symbolic heap definition to support partial data structures, e.g., list segments, which are indispensable to describe loop invariants.

Intuitively, natural proofs summarize the program execution on a symbolic heap of bounded size, on which the correctness can be thoroughly checked. Now let us graphically show a symbolic heap and a mapping from a concrete heap to it through our running example. Consider the synthesized program and proof terms as in Figure 2b and how to prove its correctness. Each basic block of `srtl_insert_iter` will be

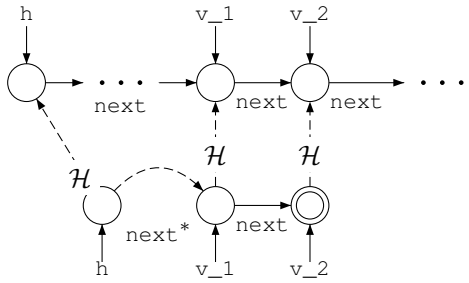

Figure 5: Concrete Heap (above) vs. Symbolic Heap (below)

first standardly converted to a verification condition; now let us focus on the loop body and prove that the loop invariant is indeed an invariant using natural proofs.

Before an iteration, the shape of the heap is shown as the upper part of Figure 5: there is a sorted list starting from `h`, and it contains locations pointed by `v_1` and its successor `v_2`. While the size of the heap is unbounded, we can summarize it using a 3-node symbolic heap (see the lower part of Figure 5). There exists a correspondence $\mathcal{H}$ between the symbolic heap and the concrete heap, which is shown in Figure 5 as well. Intuitively, $\mathcal{H}$ is a homomorphism mapping each node of the symbolic heap to a node in the concrete heap. The dotted edge labeled `next`* summarizes a list segment from `h` to `v_1`.

Now the two statements synthesized as the loop body can be considered as a sequence of manipulations on symbolic heap as shown in Figure 6. As illustrated before, the symbolic heap at the top of the figure symbolically represents an arbitrary list satisfying the loop invariant as well as the loop condition. Then the iteration can be simulated by concretizing `v_2`, and updating both `v_1` and `v_2`. As the symbolic heap is of bounded size, it can be mechanically checked that no matter how `v_2` is concretized (points to *nil* or not) and how the rest of the list is formed, the loop invariant can *always* be maintained, and the ranking function $len^*(\texttt{v\_2})$ *always* decreases till 0. This finite model checking implies that the loop invariant is preserved on any of the resulting symbolic heaps.

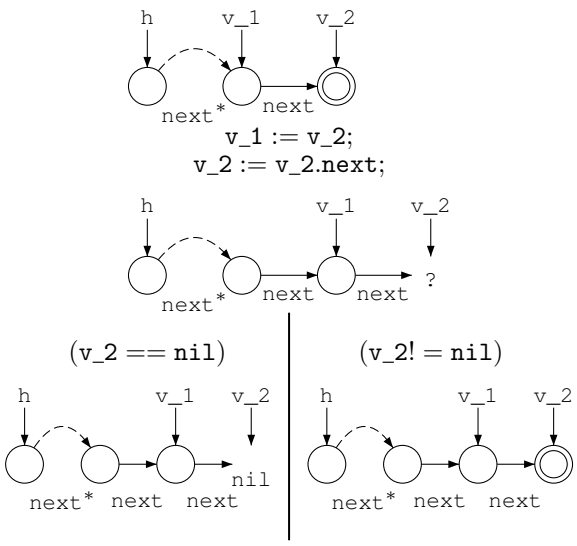

Figure 6: Nondeterministic symbolic execution of a loop iteration in `srtl_reverse_iter`

4. Reducing to A Natural Synthesis Problem

In this section, we elaborate the main step of natural synthesis: soundly reduce the unbounded synthesis problem raised from a ImpSynt template to a bounded synthesis problem. We call the problem after reduction a *natural synthesis problem*, as the reduction essentially encodes the gist of natural proofs, so that the soundness of the reduction is guaranteed. In other words, When a valid solution is found for the natural synthesis problem, it is also a valid solution for the original ImpSynt synthesis problem, guaranteed by a natural proof. The reduction is done by defining an abstract semantics of the ImpSynt program based on symbolic heap. More specifically, the abstract semantics consists of two parts: a symbolic execution of Imp programs w.r.t. symbolic heaps, and an interpreter of Dryad^{FO} formulas on symbolic heaps. In this section, we first formulate the ImpSynt synthesis task, i.e., what the input of the synthesis problem is and what the expected output is. Then we present the symbolic execution of Imp and the symbolic Dryad^{FO} interpreter, respectively. Finally we formally build the reduction and show it is sound.

4.1 The ImpSynt Synthesis Problem

As mentioned in Section 2, ImpSynt supports `requires`, `ensures`, `invariant` and `decreases` as modular contracts with Dryad^{FO} formulas. As these claims should talk about properties of the heap at the function entry as well as the current-state. we allow combining terms and formulas obtained from the function entry and the current-heap. For example, in our running example, the postcondition contains literals like $\mathit{len}^*(ret) = \texttt{old}(\mathit{len}^*(h)) + 1$, which can be interpreted as: evaluate $\mathit{len}^*(ret)$ on the current heap, and evaluate $\mathit{len}^*(h)$ on the initial heap at function entry, and the two integer values are equal. We leave the more formally defined semantics in Appendix A.

The semantics of other statements in Imp is just the ordinary heap manipulations that one can expect for each statement. When an unexpected situation is encountered, the current statement will simply abort, making the program invalid immediately. In particular, for the `malloc()` operation, we assume it always succeeds, but in reality, when the Imp program is executed in real world, e.g., with a C compiler, it may encounter the out-of-memory error, which is not considered in this paper. The concrete small-step operational semantics of Imp can be found in Appendix C, where we also formally define the program validity of Imp programs. Then the ImpSynt synthesis problem can be formally stated as: given an ImpSynt program `sk`, find a solution to fill all the holes in `sk` such that the complete program is valid.

4.2 Symbolic Execution of Imp

In the previous section, we have illustrated a symbolic execution on a symbolic heap through Figure 6. The symbolic execution in this paper is mostly similar to the one presented in [14]. Conceptually, it summarizes all possible executions on all concrete heaps corresponding to the symbolic heap. In particular, when a statement manipulates on a symbolic node or a symbolic edge, we need to unfold this node/edge. For example, in Figure 6, to symbolically execute `v_2 := v_2.next`, we first unfold the symbolic node pointed by `v_2`. Note that there are two exclusive cases: the `h.next` can be either *nil*, or another non-nil record in the heap. The two cases leads to two *nondeterministic* choices of the execution, represented as the two branches in Figure 6.[2]

The nondeterministic symbolic execution is not defined for while-loops, i.e., one can simulate the real execution of any basic-free block in Imp . The formal semantics for each kind of statements can be found in Appendix C. It can be verified that the symbolic execution mimics real executions on concrete heaps precisely.

A basic block may also contain function calls, which need to be handled carefully. We assume an invoked function f is always equipped with a precondition and a postcondition, and there is a symbolic interpreter (will be explained shortly) to check them. Then to symbolically call f, we first check if the precondition of f is satisfied. If not, the symbolic execution immediately aborts; otherwise all nodes reachable from any of f's `loc` arguments will be deleted, and a new portion of the symbolic heap will be *nondeterministically* generated, which can be potentially overlapped with the deleted portion. The updated symbolic heap needs to satisfy the postcondition of f, or aborts immediately. It can be verified that this nondeterministic function call overapproximates all possible real function calls. In other words, with function calls, the soundness of the symbolic execution still holds.

[2] Another source of nondeterminism is the value of `h.key`, which is not reflected in Figure 6.

Note that each nondeterministic statement execution (including function calls) yields only finite number of heap shapes. Therefore the program execution on unbounded-size heaps can be summarized on a bounded number of finite symbolic heaps (modulo unbounded integer-field values).

4.3 Symbolically interpreting DRYAD^{FO} Formulas

To verify an IMP program modularly, one needs to check the validity of each Hoare-triple built from each basic block. To this end, we symbolically interpret DRYAD^{FO} formulas on symbolic heaps. The interpretation is partial because a symbolic node may represent an arbitrary tree, and a symbolic edge may represent an arbitrary list segment; therefore the values of recursive definitions on it cannot be determined. In that case, we assume the function *Orcl* can interpret it correctly in any context.

The details of the interpretation algorithm can be found in Appendix C. The algorithm has encoded the gist of natural proofs: the oracle function *Orcl* abandons the semantics of recursive definitions on symbolic nodes, but the lost precision is recovered by making sure that the semantics is at least correctly described on concrete nodes. When *Orcl* is allowed to return *arbitrary* value, i.e., becomes uninterpreted, our interpreter covers all possible interpretations, and can be used to check specifications written in DRYAD^{FO} in a sound but incomplete manner.

4.4 Reduction

We have shown IMP 's symbolic semantics and DRYAD^{FO}'s symbolic interpreter based on symbolic heaps; now to define the natural synthesis problem, we also need to show all possible program states can be covered by a bounded number of symbolic heaps. Thus we introduce *normal formulas* and *normal programs*, which are defined in Appendix C. Intuitively, a formula φ is normal if any `loc` variable in φ is the head of a tree or a partial tree; a program is normal if all formulas involved in its contract are normal. For the rest of the paper, we assume all formulas are normal.

Similar to the program validity based on concrete semantics, we also introduce the notion of *symbolic validity*, which is also defined in Appendix C. Then the natural synthesis problem is to synthesize a *normal and symbolically valid* IMP program. The reduction from an IMP synthesis problem to its corresponding natural synthesis problem is sound:

Theorem 4.1. *Given an* IMPSYNT *program* `sk`*, if there exists a way to fill all the holes to form a* IMP *program* `S` *which is normal and symbolically valid, then* S *is also valid.*

Proof. See Appendix C. □

5. Formulating the Natural Synthesis Problem

In the previous section, we have shown how to (soundly) reduce the IMPSYNT synthesis problem to a natural synthesis problem, more specifically, synthesizing a normal and symbolically valid program. The next step of natural synthesis is to encode the natural synthesis problem as a logic query, which can be solved using state-of-the-art synthesis engines, such as SKETCH [21] or ROSETTE [24]. Now we show how to encode the natural synthesis problem to a *synthesis condition* in common theories including arrays and uninterpreted functions. Our ideas are as follows: a) represent the symbolic heap using arrays of bounded size; then b) encode each kind of statement as a sequence of manipulations on the arrays, according to their symbolic semantics; c) interpret DRYAD^{FO} formulas on arrays that encode the heap; finally d) encode the search space in the IMPSYNT program using generator functions. Now let us explain each of these steps.

Symbolic heaps as arrays: We encode symbolic heaps using arrays, which is a first-class type with built-in support in most modern solvers. We assume fixed bounds for the size of symbolic heaps, the number of location/integer variables, and the number of pointer/data fields.[3] We represent these bounds as $\mathtt{B_{HEAP}}$, $\mathtt{B_{LVAR}}$, $\mathtt{B_{IVAR}}$, $\mathtt{B_{DIR}}$ and $\mathtt{B_{DATA}}$, respectively, and declare the following arrays: *locvars*, *intvars*, *active*, *symbolic*, *semisym*, *dirs* and *data*. Their names have explained roughly what they represent. For example, *symbolic* indicates if a location is symbolic. Their precise meanings and the bound constraints required on them can be found in Appendix D.

Encoding statements and DRYAD^{FO} logic: We can simulate the symbolic execution presented in Section 4.2 as manipulations on the arrays defined above. We encode symbolic execution for each kind of statements as an array-manipulating function. Most statements are deterministic and the encoding is quite straightforward. For example, a pointer field mutation statement $u.dir := v$ is encoded as a function that accepts three id's, representing u, *dir* and v, and modifies the array *dirs* appropriately to reflect the field mutation. Specifically, for function calls, we create a special $havoc$ function to nondeterministically (will explain shortly) update all reachable locations from the call arguments. We presume that assumptions and assertions are supported in the synthesis engine, then for each function call, the simulation consists of four steps: 1) assert its precondition; 2) snapshot the current configuration in a set of reference arrays; 3) call the $havoc$ function; 4) assume its precondition satisfied (old-formulas referred to the reference arrays). We leave the detailed encoding in Appendix D.

After encoding of DRYAD^{FO} logic is also shown in Appendix D. As we have shown in Section 4.3, the symbolic interpreter for natural synthesis relies on an oracle *Orcl*. In principle, the encoding of DRYAD^{FO} is mostly an implementation of that interpreter modulo the *Orcl* function. For every inquiry $Orcl(R^*, n)$ to the oracle, we replace it with a

[3] As mentioned before, the symbolic heap is always of bounded size. Hence this assumption does not necessarily affect the soundness of the problem, as long as the bounds are sufficiently large.

special function $sym\_R(n)$, which guarantees to explore all possible values might be returned by the inquiry.

Nondeterministic choices as uninterpreted functions: Nondeterminism is involved in both the symbolic execution and the DRYAD^{FO} interpreter. In particular, we have assumed sym_R for each recursive definition R^*. We encode these functions using *uninterpreted functions*. An uninterpreted function can be interpreted *arbitrarily*; so from the perspective of synthesis, the synthesizer should produce a program that meets the specification no matter what the uninterpreted functions output. Therefore all possible nondeterministic executions will be considered.

Besides regular inputs like the id's of nodes and fields, these uninterpreted functions also need to accept as input one more dimension for time, because a single id may represent different nodes along with the symbolic execution. For example, consider symbolic node, being concretized and then freed, and again being allocated due to the unfolding of another symbolic node. The technical details about these uninterpreted functions can be found in Appendix D.

Encoding unknown holes in* IMPSYNT *: Recall that our goal is to encode a IMPSYNT synthesis problem. We have shown how to encode statements in SKETCH and formulas specified in DRYAD^{FO}; what remain are the unknown holes allowed in IMPSYNT , including unknown variables or fields (`??`), unknown conditions (`cond(`C`)`), unknown values (`val(`C`)`), unknown statements (`stmt(`C`)`), unknown conjunctions (`conj(`C`)`), and unknown terms (`exp(`C`)`). The variable hole `??` can be simply encoded as an unknown integer for the variable's id. Other kinds of unknowns are encoded using *generator functions*, which are commonly supported by syntax guided program synthesizers. For example, a hole `conj(`C`)` can be translated to a call to a generator function `conj` with input C.[4] The encoding for each kind of holes can be found in Appendix D.

So far we have systematically encoded the whole natural synthesis problem, including symbolic execution, DRYAD^{FO} formula interpretation and program holes. Given an IMPSYNT program `sk`, let us denote the encoded synthesis condition as *encode*(`sk`). The following theorem shows that each solution to the synthesis condition corresponds to a normal and symbolically valid solution to the original synthesis problem, and thus also a valid solution for real execution, by Theorem 4.1.

Theorem 5.1. *Let* `sk` *be an* IMPSYNT *program, then* `sk` *has a normal and symbolically valid solution if and only if* $encode(sk)$ *has a solution.*

Proof. See Appendix D. □

[4] For each conjunction hole `conj(`C`)` occurred in a loop invariant, we also place assertions in the corresponding generator function to guarantee that the generated conjunction is normal, and therefore any synthesized program will be normal.

6. Experiments

After encoding the natural synthesis problem as a synthesis condition, we pick SKETCH [21], a state-of-the-art inductive synthesizer, as the backend synthesis engine of our natural synthesis-based program synthesizer. In this section, we first introduce SKETCH and some implementation-related details of our system, then evaluate our natural synthesis approach through a set of experiments.

6.1 SKETCH and Encoding Optimizations

SKETCH [21] is a synthesis-enabled language syntactically similar to C. It supports all the features required by our synthesis condition set forth in the previous section: arithmetic, arrays, uninterpreted functions, generator functions, etc. Each basic block can be written as a harness function with assumptions and assertions. The synthesized program must guarantee the absence of assertion failures, unless a prior assumption is not satisfied.

SKETCH finitizes the synthesis problem by inlining function calls and unrolling loops finitely, and formulate the problem as a boolean formula: $\exists\phi.\ \forall\sigma.\ Q(\phi,\ \sigma)$, 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.

SKETCH can effectively solve the synthesis problem described by $encode(sk)$, modulo the completeness to integer arithmetic [21], as SKETCH finitizes the integer domain by representing them using fixed number of bits. However, once a solution is found, SKETCH invokes an off-the-shelf SMT solver to verify the solution's validity. Therefore, Theorem 5.1 implies that we can algorithmically find normal and symbolically valid solutions for IMPSYNT sketches. Then by Theorem 4.1, the solution is indeed valid. Thus, we have built a procedure to produce complete IMP programs from IMPSYNT sketches, with its correctness guaranteed by natural proofs.

To tackle the scalability, we manage to control the size of the encoded synthesis problem with several optimization techniques. For instance, to keep the heap size bound small yet avoid the out-of-memory error, we mechanically compute the "malloc budget", the maximum number of malloc needed in the program, and exclude those programs involving too many malloc's or unfoldings. Moreover, several program-independent axioms are necessary in proving many properties, e.g., for any location l, $len^*(l) \geq 0$, $min^*(l) \leq max^*(l)$, and $sorted\_l^*(l) = sorted\_lseg^*(l, nil)$. We encode these axioms as assumptions in SKETCH.

We also find some other assumptions that are very helpful in reducing the search space. For example, we break the redundancy caused by the symmetry in the variable store by assigning independent variables to fixed locations in the heap. We also assume the ranking function is always in

Example	Cond/Loop synthesized?	Hole size	Num. of iteration	Time
sll_max_rec	Y / N	112	28	16s
sll_max_iter	Y / Y	95	33	19s
sll_min_rec	Y / N	112	28	16s
sll_min_iter	Y / Y	95	26	15s
sll_len_rec	Y / N	102	11	7s
sll_len_iter	Y / Y	95	29	14s
srtl_prepend	N / N	10	3	4s
srtl_reverse_iter	Y / Y	154	156	75m43s
srtl_insert_rec	Y / N	59	17	21s
srtl_insert_iter	Y / Y	141	75	18m14s
insertion_sort_rec	Y / N	44	16	1m34s
bst_left_rotate	N / N	25	3	17s
bst_right_rotate	N / N	25	4	14s
bst_insert_rec	Y / N	72	17	5m43s
bst_del_root_rec	Y / N	53	32	40m16s

Figure 7: Experimental results of synthesizing provably-correct data-structure manipulations

the form $f^*(v)$, where f^* is a recursive function and v is a mutable variable involved in the loop.

6.2 Experimental Evaluation

To demonstrate the effectiveness of our natural synthesis approach and our synthesizer, we conducted experiments on synthesizing 15 data-structure manipulations, of which six manipulate singly-linked lists, five manipulate sorted lists and four manipulate standard binary-search tree trees. For each example, we wrote a full functional specification in terms of pre-/post-conditions, with a bare-bones program template at minimal amount of burden: only top-level branches, loops and function calls, with all the implementation details unknown. We didn't even write a single line of statement in these templates. These templates were written in IMPSYNT , and our natural synthesizer automatically encoded the natural synthesis condition presented in Section 5 to SKETCH.

The first set of examples (with prefix `sll_`) synthesizes standard manipulations on singly-linked lists. The names are fairly self-explanatory: the middle part (`max`, `min` or `len`) indicates the aim is to find the max key, the min key or the length of the list; the suffix (`_rec` or `_iter`) indicates the expected implementation should be a recursive function call or a while-loop.

The second set of examples involve sorted lists. The example `srtl_prepend` accepts a sorted singly-linked list L and a key k not greater than any element in the list, and is expected to return a sorted list with k as the head and the the old list as the tail. `srtl_reverse_rec` and `srtl_reverse_iter` take as input a sorted list, and expects as output a reverse-sorted list. As our running examples, `srtl_insert_rec` and `srtl_insert_iter` have been elaborated in previous sections. The example `insertion_sort` is expected to implement the insertion sort algorithm in a recursive manner. It is noteworthy that this example is synthesized modularly: in addition to recursively calling itself, this program template also calls an arbitrary `srtl_insert` implementation to do insertion. All these examples' postconditions specify the result list's length, min and max should be as expected.

The last set of examples synthesizes standard manipulations on binary search trees. The example `bst_left_rotate` (`bst_right_rotate`) is expected to perform the left (right) rotate manipulation on a BST with root r, ensuring the output is still a BST with original r's right (left) c hild as the new root. The example `bst_insert_rec` recursively insert a new key into a BST, and ensures what returned is still a BST. The example `bst_del_root_rec` expects a non-empty BST as input, and recursively deletes the root of the tree, and ensures that what returned is still a BST. All these examples' postconditions specify the result tree's min and max, height and size should be as expected.

The experiments were conducted on a 2.8GHz, 16GB laptop running OS X 10.9 and SKETCH 1.7.0. We summarize our experimental results in Figure 7. For each example, we report whether the branch conditions that determine the control flow were synthesized, whether the loop condition, loop invariants and the ranking functions were synthesized, the hole size (in bits) of the internal model in SKETCH which represents the search space of the synthesis problem, the number of iterations performed by the CEGIS engine, and the total time spent by SKETCH. The synthesis results are encouraging: our system successfully produced verified implementations as well as necessary loop invariants and ranking functions. 10 out of 15 programs were synthesized and verified within one minute; other programs were synthesized and verified in longer but still reasonable time, taking into account the inherent difficulty of the synthesis problems they represent. All the synthesized implementations are nontrivial and similar to the ones an ordinary programmer may write. Each basic block in the synthesized implementation has up to 6 atomic statements; and the synthesized loop invariant consists of up to 8 conjuncts. To our knowledge, this is the first program synthesizer that can efficiently produce provably-correct imperative data-structure manipulations, with minimal user inputs and such rich functional correctness specification.

7. Related Work

Conceptually, to build provably-correct programs from rich specifications, three different tasks need to be done:

a) produce a complete program (Program Synthesis);

b) produce additional annotations, including loop invariants, ranking functions and proof tactic advice for proving program correctness (Annotation Synthesis); and

c) generate verification conditions using logical semantics of the programming language, and check their validity (Program Verification).

Existing approaches can be roughly divided into two classes, according to how the above tasks are accomplished. *Deductive synthesis* tackles all of the above tasks in tandem. As a traditional approach, it has received intensive research efforts since the late 1970s [6, 27] to recent years [4]. Following the correct-by-construction methodology, this approach derives a correct program from a set of inference rules. In recent years, due to the significant progress in automated constraint solving, *inductive synthesis* has also become feasible. Unlike deductive synthesis, this approach separates program synthesis and verification explicitly, by an "enumerate-and-verify" loop [21, 24]. In this section, we only discuss the work that is closest to ours.

Deductive synthesis: Several systems have been developed for deductive synthesis of data-structure manipulations, but they aim to achieve their goals in an interactive fashion. Kneuss et al. [10] present a system that automatically builds recursive functional programs from algebraic inductive specifications. One the one hand, their system produces recursive function calls and conditionals, i.e., they do not synthesize imperative programs with destructive heap updates; on the other hand, not all programs synthesized by their system are automatically verified. Our work aims at synthesizing imperative program manipulating on mutable heaps, which is more challenging, as the destructive updates, loop invariants and ranking functions look very different from the program specification. Moreover, we always guarantee the synthesized programs are provably correct, and no additional verification required.

A more recent work by Delaware [4] also stems from the tradition of deductive synthesis and synthesizes SQL-like operations. When *imperative* manipulations with destructive updates and loops are desirable, their system tends to require implementation-related guidance from the user. In contrast, our system only requires a bare-bones control flow of the desired program, and leaves all the implementation details automatically synthesized in a push-button style.

Data representation synthesis [7] efficiently builds high-level data relations from low-level, primitive data-structure manipulations, which provably satisfy the relational specification. Our system allows expressing more general properties in DRYAD$^{\text{FO}}$, and synthesizing more general heap manipulations as well as auxiliary inductive invariants and ranking functions.

The closest work to ours is *proof-theoretic synthesis* [22]. This work was proposed to automate the correct-by-construction approach; the main insight behind this work is to encode all the three tasks to a logical query and solve it using a constraint solver. Nonetheless, proof-theoretic synthesis is not directly applicable for synthesizing data-structure manipulations. Due to the high complexity of heap structure and data reasoning, there will be no constraint solver to automatically handle those data-structure properties involved in the synthesis condition. They synthesized sorting algorithms that are implemented using arrays, instead of linked lists, as these verification tasks can be solved by their specific arithmetic solvers. Our natural synthesis approach is motivated by this restriction, and generalizes proof-theoretic synthesis. By reducing those intractable synthesis problems to natural synthesis problems, the synthesis conditions can be significantly simplified and encoded using common logic theories, so that modern synthesis engines are readily available.

Inductive synthesis: Inductive synthesis relies on a standalone solver to verify each proposed program. To ensure that the synthesize-verify loop runs efficiently, inductive synthesis based systems usually finitize both the execution paths and the program states, leaving the task b) unaddressed and the task c) unsound for data-structures of unbounded sizes. However, for some specific domains, researchers proposed abstraction-guided synthesis [20, 25, 26], which guarantees partial correctness of the synthesized program.

The insight behind abstraction-guided synthesis is similar to our natural synthesis reduction: the abstraction can be understood as a sound reduction of the synthesis problem, which also converts an unbounded synthesis problem to a bounded one. However, abstraction-based techniques are typically not powerful enough to handle very rich functional correctness desired by the programmer. For example, as the state-of-the-art, the *Storyboard* system [20] supports specification by input-output examples which can describe some global properties, but it is in general not expressive enough to express logical specification for full functional correctness. For example, the sortedness of lists cannot be specified in *Storyboard*. Moreover, the means to guarantee termination in Storyboard is not applicable to more general data-structure manipulations. We believe the same problems remain if one tries to couple inductive synthesis with other abstractions, e.g. TVLA [11, 12], as they check partial correctness only and cannot handle any arithmetic properties, e.g., the length of a list. In contrast, our natural synthesis approach, for the first time, integrates inductive synthesis with a full-fledged arithmetic logic and guarantees total correctness.

Heap verification: There is a rich literature in verification of heap properties and data structures. Our work is based on natural proofs [14, 18] which have been elaborated in Section 3. A standalone natural-proof verifier [17] that requires only pre-/post-conditions and loop invariants were also built based on VCC [3]. Similar proof tactics by unfolding recursive definitions were also explored in [1, 23]. Other program verifiers [2, 9] based on separation logic [16, 19] are also available, but usually require certain level of guidance from the user. Another group of related work develops decision procedures, based on both separation logic and non-separation logic, among which [8] and [13] are very powerful.

Appendices

See the supplemental material submitted with this paper.

References

[1] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, pages 1006–1036, 2012.

[2] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI '11*, pages 234–245, 2011.

[3] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[4] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *POPL'15*, pages 689–700. ACM, 2015.

[5] A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In *OOPSLA'14*, pages 709–725. ACM, 2014.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[7] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI'11*, pages 38–49. ACM, 2011.

[8] R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE-24*, pages 21–38, 2013.

[9] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: a powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*, pages 41–55, 2011.

[10] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA'13*, pages 407–426. ACM, 2013.

[11] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS '00*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000. doi: 10.1007/978-3-540-45099-3_15.

[12] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00*, pages 26–38. ACM, 2000. doi: 10.1145/347324.348031.

[13] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL'11*, pages 611–622. ACM, 2011.

[14] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL'12*, pages 123–136. ACM, 2012.

[15] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[16] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[17] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *PLDI'14*, pages 440–451. ACM, 2014.

[18] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI '13*, pages 231–242. ACM, 2013.

[19] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE-CS, 2002.

[20] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *ESEC/FSE'11*, pages 289–299. ACM, 2011.

[21] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS'06*, pages 404–415. ACM, 2006.

[22] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL'10*, pages 313–326. ACM, 2010.

[23] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, pages 298–315, 2011.

[24] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI'14*, pages 530–541. ACM, 2014.

[25] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI'08*, pages 125–135. ACM, 2008.

[26] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky. CGC-Explorer: A semi-automated search procedure for provably correct concurrent collectors. In *PLDI'07*, pages 456–467. ACM, 2007.

[27] N. Wirth. *Systematic Programming: An Introduction*. Prentice Hall, 1973.