Natural Synthesis of Provably-Correct Data-Structure Manipulations

XIAOKANG QIU, Purdue University, USA
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology, USA

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, their functional correctness and their termination in tandem from bare-bones control flow skeletons.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; *Source code generation*; *Formal software verification*; • **Theory of computation** → *Logic and verification*; *Program verification*; Hoare logic; Invariants; • **General and reference** → *Verification*;

Additional Key Words and Phrases: Data structures, Natural proofs, Synthesis conditions

ACM Reference Format:
Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural Synthesis of Provably-Correct Data-Structure Manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 65 (October 2017), 28 pages.
https://doi.org/10.1145/3133889

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

Xiaokang Qiu: xkqiu@purdue.edu
School of Electrical and Computer Engineering, Purdue University
465 Northwestern Ave, West Lafayette, IN 47907, USA.
Armando Solar-Lezama: asolar@csail.mit.edu
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
77 Massachusetts Ave, Cambridge, MA 02139, USA.

2475-1421/2017/10-ART65
https://doi.org/10.1145/3133889

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 [Manna and Waldinger 1979], this goal typically can only be achieved for finite programs [Solar-Lezama et al. 2006; Torlak and Bodik 2014] or with user interaction [Delaware et al. 2015; Kneuss et al. 2013].

In recent years, due to the significant progress in automated constraint solving, *proof-theoretic synthesis* [Srivastava et al. 2010] 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 [Chlipala 2011; Iosif et al. 2013; Jacobs et al. 2011; Madhusudan et al. 2011] 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 [Delaware et al. 2015; Kneuss et al. 2013] or support only weak specifications in the form of input/output samples [Singh and Solar-Lezama 2011].

This paper presents *natural synthesis*, a novel approach to automated program synthesis, which aims to generalize proof-theoretic synthesis to handle more sophisticated logic theories. In a nutshell, natural synthesis is an approach to synthesizing programs whose correctness can be proved using a specific class of proofs called *natural proofs* [Desai et al. 2014; Madhusudan et al. 2012; Pek et al. 2014; Qiu et al. 2013]. 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 is 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 [Solar-Lezama et al. 2006] or Rosette [Torlak and Bodik 2014], resulting in a fully automatic synthesis algorithm.

This paper focuses on synthesizing imperative provably-correct data-structure manipulations over dynamically-allocated heap. Following the natural synthesis methodology, we develop a syntax-guided program synthesizer that can produce imperative, provably-correct data-structure manipulations. The input to the synthesizer is a program template written in ImpSynt , a heap-manipulating, synthesis-enabled language. ImpSynt allows the user to describe merely 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

$$
\begin{array}{rcl}
\text{Program } p & ::= & \epsilon \mid p\,;\,p \mid T\ f(arg)\ \{\textbf{requires}\ \varphi\,;\ \textbf{ensures}\ \varphi\,;\ \textbf{decreases}\ it\,;\ blk\} \\
\text{Arguments } arg & ::= & \epsilon \mid arg\,,\,arg \mid \textbf{loc}\ u \mid \textbf{int}\ j \\
\text{Block } blk & ::= & \epsilon \mid blk\,;\,blk \mid \textbf{if}\ (cnd)\ \{blk\}\ \textbf{else}\ \{blk\} \\
 & & \mid \textbf{while}\ (cnd)\ \{\textbf{invariant}\ \varphi\,;\textbf{decreases}\ it\,;\,blk\} \\
\text{Cond. } cnd & ::= & exp == var \mid exp < var \mid !cnd \mid cnd\ \&\&\ cnd \\
\text{Stmt. } st & ::= & T\ var := exp \mid var.fld := var \mid \textbf{loc}\ u := \textbf{malloc}() \mid \textbf{free}(u) \\
 & & \mid \textbf{return}\ var \mid T\ var := f(var_1, \ldots, var_n)
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\text{Type } T & ::= & \text{loc} \mid \text{int} & \text{Expr. } exp & ::= & var \mid var.fld \mid \textbf{nil} \mid C \\
\text{Var. } var & ::= & u \mid \ldots \mid j \mid \ldots & & & \mid exp - exp \mid exp - exp \\
\text{Field } fld & ::= & dir \mid \ldots \mid df \mid \ldots & & & \mid cond\,?\,exp : exp
\end{array}
$$

f : Function Name u : Loc Var. j : Int Var. $dir \in Dir$ $df \in DF$ C : Int Const
it : Int. Term in DRYAD^{FO} φ : DRYAD^{FO} Formula (allowing the old construct)

Fig. 1. Grammar of IMP

invariants and ranking functions. The salient features of the program synthesizer set forth in this paper include: a) supports rich functional specifications written in an expressive program logic; b) synthesizes desired implementations (e.g., recursive or iterative) from user's high-level guidance; c) guarantees very rich functional correctness and termination by synthesizing implementations and proof artifacts in tandem; d) produces automatically 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 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 reduction to a natural synthesis problem as the main idea of natural synthesis. 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. Section 7 discusses related work that is closest to ours.

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. IMP also allows the programmer to specify preconditions, postconditions, loop invariants and ranking functions in DRYAD^{FO}, a dialect of

the Dryad logic [Madhusudan et al. 2012]. 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 *Dir* and a set of data fields *DF*. However, the programmer can in practice define multiple struct types. The formal semantics of Imp will be discussed in the following sections.

Our system allows the programmer to provide a program sketch 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**()/**val**() represents an unknown condition/expression;
(3) **stmt**(C) represents an arbitrary loop-free code snippet which 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/integer-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 **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$_k$:= ??;

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

where **preserves_old**() is a conjunction of conditions for preserving old values: for each term old($f^*(t)$) appeared in the final postcondition, we assume the old value preserved during the loop as a term **exp**(2), i.e., the loop invariant contains a formula old($f^*(t)$)=**exp**(2). As a simpler alternative, **simple–loop**(V, N) disallows the destructive updates to lv$_1$ through lv$_V$ at the end of the loop body. [1]

To write a program template, ImpSynt requires the user to provide **requires** and **ensures** formulas as pre-/post-conditions for each function. ImpSynt also expects the user to convey her high level design decisions, in terms of a program template describing high-level control flow and 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

[1] V and N are usually very small, as they 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. The programmer may estimate a reasonable number, or start from a small number and increment it if the synthesizer fails to find a solution.

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

```
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()) { // 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: A 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 := h;
    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 program in Imp

Fig. 2. Synthesizing a sorted-list-insertion program (iterative implementation)

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, to make sure the input key is indeed inserted, 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^{FO}. The formulas involve several recursively-defined predicates and functions: $sorted\_l^*$, len^*, min^* and max^*, which are pre-defined in our system and will be explained in the next section. [2]

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 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.

```
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()) {   // 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 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 := h;
    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 program in IMP

Fig. 3. Synthesizing a sorted-list-insertion program (recursive implementation)

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. Moreover, the two parameters 2 and 0 are determined by the programmer based on her understanding of the programming task: 2 loc variables should be maintained throughout the loop to find the two locations right before and after the place of insertion; and obviously none integer variable needs to be updated within the loop. Note that the numbers involved in the template are given by the user as an (over)approximated number of involved variables; they are not necessarily given precisely. Actually in this particular example, slightly larger

[2] In general, the programmer can define arbitrary recursive predicate or function for her custom data structures and programming tasks.

numbers are still manageable in our system and can yield the same results. 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, and the return variable. More importantly, our system also finds a proof of the synthesized program's *total correctness*, by generating an inductive 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. Notice that our system also synthesized four statements from a single **stmt**(1) template, as the four statements can be regarded as a single malloc-and-initialize super-statement. Most impressively, 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. 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. For the trivial case, the synthesized code is identical to the iterative version. For the non-trivial case, our system successfully finds the successor of h as a sublist for recursive call, and update the next field of the root with the newly constructed sublist from the call. 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^{FO}

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 paper, we adopt the Dryad logic and the natural proof methodology presented in [Madhusudan et al. 2012] and adapt them for the purpose of synthesis. We first give a summary of the existing logic Dryad^{FO}, the first-order fragment of Dryad. We refer the reader to [Madhusudan et al. 2012] for more details. We then show how we generalize the natural proof tactics to handle partial data structures and loops, and intuitively, how natural proofs work through our running example.

$dir \in Loc$ Fields $\quad i^* : Loc \to Int \quad x, y \in Loc$ Variables $\quad c : Int$ Constant
$f \in Int$ Fields $\quad p^* : Loc \to \{\text{true}, \text{false}\} \quad j \in Int$ Variables $\quad q \in Boolean$ Variables

$$Loc \text{ Term: } lt, lt_1, lt_2 \ldots ::= x \mid \text{nil} \mid lt.dir$$
$$Int \text{ Term: } it, it_1, it_2, \ldots ::= c \mid j \mid lt.f \mid i^*(lt) \mid i^*(lt, y) \mid it_1 + it_2 \mid it_1 - it_2 \mid \text{ite}(\varphi, it_1, it_2)$$

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

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

Fig. 4. Syntax of DRYAD$^{\text{FO}}$ Logic

3.1 Preliminaries

The DRYAD$^{\text{FO}}$ logic is designed to specify heap structure and data properties using two sorts: *Loc* for locations in the heap and *Int* for integers. It is quantifier-free but allows recursively-defined integer functions and predicates.

***Syntax of* DRYAD$^{\boldsymbol{FO}}$.** The syntax of DRYAD$^{\text{FO}}$ is shown in Figure 4. A unary recursive function is defined using the syntax $i^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil},\ i_{base},\ i_{ind})$, where i_{base} and i_{ind} are themselves terms that stand for what i^* evaluates to when $x = \text{nil}$ (the base-case) and when $x \neq \text{nil}$ (the inductive step), respectively. For example, the length of a list can be recursively defined in DRYAD$^{\text{FO}}$: $len^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, 0, len^*(x.\text{next}) + 1)$; and the definition of $sorted\_l^*$ relies on min^*: $sorted\_l^*(x) \stackrel{def}{=} \text{ite}\big(x = \text{nil}, \text{true},\ sorted\_l^*(x.\text{next}) \wedge x.\text{key} \le min^*(x.\text{next})\big)$. Other predicates and functions can also be easily defined. DRYAD$^{\text{FO}}$ also extends original DRYAD with 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 becomes $x = y$.

For technical reasons, there is a set of syntactic restrictions on these definitions. To guarantee a well-defined function, DRYAD$^{\text{FO}}$ places different syntactic restrictions on these two terms:

- i_{base} has no free variables and hence evaluates to a fixed integer value.
- i_{ind} only has x as a free variable. Furthermore, x can only be dereferenced at most once ($x.dir$ or $x.f$).

A recursive predicate p^* is defined with similar syntax and restrictions. Intuitively, when $x \neq \texttt{nil}$, it evaluates to a function that is defined recursively using properties of the location x, including pointer/data fields of x, and these properties may in turn involve p^* itself and other recursively defined predicate/functions.

We assume that the inductive definitions are not *circular*. Formally, let *Def* be a set of definitions and consider a recursive definition of a function f^* in *Def*. Define the sequence $\psi_0, \psi_1, \ldots$ as follows. Set $\psi_0 = f^*(x)$. Obtain ψ_{i+1} by replacing every occurrence of $g^*(x)$ in ψ_i by $g_{ind}(x)$, where g is any recursively defined function in *Def*. We require that this sequence eventually stabilizes (i.e. there is a k such that $\psi_k = \psi_{k+1}$). Intuitively, we require that the definition of $f^*(x)$ be rewritable into a formula that does not refer to a recursive definition of x (by getting rewritten to properties of its descendants). We require that every definition in *Def* have the above property.

***Semantics of* DRYAD$^{\boldsymbol{FO}}$.** The semantics of DRYAD$^{\text{FO}}$ is formally 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. Formally, a concrete heap is defined as follows ($f : A \rightharpoonup B$ denotes a partial function from A to B):

Definition 3.1 (Concrete Heap). A concrete heap over a set of pointer-fields *Dir*, a set of data-fields *DF*, and a set of program variables *PV* is a tuple (N, L, pf, df, pv) where:

- N is a finite set of locations, which includes a special location *nil*, representing the null pointer;
- $L \subseteq N$ represents the allocated locations of the heap (assuming *nil* is always in L);
- $pf : (L \setminus \{nil\}) \times Dir \to N$ is a function that maps every non-nil location l and every pointer field *dir* to the location pointed by the *dir* field of l;
- $df : (L \setminus \{nil\}) \times DF \to \mathbb{Z}$ is a function that maps every non-nil location l and every data field *dir* to the location pointed by the *dir* field of l;
- $pv : PV \rightharpoonup N \cup \mathbb{Z}$ is a partial function mapping program variables to locations or integers, depending on the type of the variable. □

A Dryad$^{\text{FO}}$ formula can be interpreted on a concrete heap (N, L, pf, df, pv) if the variables occurred in the formulas are all from pv. Each variable is interpreted according to the function pv. Each term evaluates to either a normal value of the corresponding type, or to `undef`. For a *Loc* term of the form $lt.dir$, if lt is evaluated to a normal node $n \in L \setminus \{nil\}$, then $lt.dir$ is evaluated by looking up $pf(n, dir)$; otherwise, the field accessing fails and the term simply evaluates to `undef`.

The semantics of recursive definitions is more noteworthy. Note that the structure part of a concrete heap (N, L, pf, df, pv) can be viewed as a directed graph: N is the nodes and pf is the directed edges between nodes with labels from *Dir*. Then to guarantee the recursive definitions can be eventually reduced to the base case, Dryad$^{\text{FO}}$ requires the portion of the heap used in the evaluation to form a tree. Concretely, let a recursive definition of the form $i^*(lt)$ or $p^*(lt)$ is defined with respect to pointer fields in *Dir*, it will evaluate to `undef` if lt evaluates to `undef`, or the subgraph reachable from lt via *Dir* forms a tree; otherwise it will be evaluated recursively using its recursive definition.

Similarly, binary definitions $i^*(lt, lt')$ or $p^*(lt, lt')$ is evaluated to `undef` if lt or lt' evaluates to `undef`, or the subgraph reachable from lt *without* passing through lt', forms a tree. The predicate `disjoint`(x, y) means $pv(x)$ and $pv(y)$ are the roots of two disjoint trees.

Other constructs of the logic are interpreted with the usual semantics of integers and Boolean logic, unless they contain some sub-term or sub-formula evaluating to `undef`, in which case they also evaluate to `undef`. In other word, `undef` will be propagated throughout the whole formula, and finally be replaced with `false`.

Special definitions. Note that the treeness is a first-class property in Dryad$^{\text{FO}}$ semantics, and itself can be defined recursively. Consider a special recursively defined predicate *tree* that is defined as:

$$tree^*(x) \stackrel{def}{=} \mathsf{ite}(x = \mathsf{nil}, \mathsf{true}, \mathsf{true})$$
$$tree^*(x, y) \stackrel{def}{=} \mathsf{ite}(x = y, \mathsf{true}, \mathsf{true})$$

Note that since recursively defined predicates can hold only on trees and since the above formulas vacuously hold on any tree, $tree^*(x)$ holds iff x is a root of a *Dir*-tree, and $tree^*(x, y)$ holds iff the region reachable from x without passing through y forms a *Dir*-tree.

The reachability between locations can also be defined recursively. Let u be a fixed *Loc* variable, we can define a recursive predicate $reach^*_u(x)$, indicating u is reachable from x:

$$reach^*_u(x) \stackrel{def}{=} \mathsf{ite}\Big(x = \mathsf{nil}, \mathsf{false}, x = u \ \vee \bigvee_{dir \in Dir} reach^*_u(x.dir)\Big)$$

We assume that $tree^*$ and arbitrary $reach^*_u(x)$ are always implicitly defined.

***Proof Tactics for* Dryad^{FO}.** *Natural Proofs* [Desai et al. 2014; Madhusudan et al. 2012; Pek et al. 2014; Qiu et al. 2013] is a proof methodology proposed to ease the inherent tension between the logic expressiveness and the reasoning automaticity. Two proof tactics were identified in [Madhusudan et al. 2012; Qiu et al. 2013] for automated program verification: 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. These tactics were formally described using a formalism called *symbolic heap*. Intuitively, natural proofs summarize the program execution on a symbolic heap of bounded size, on which the correctness can be thoroughly checked. In this paper, we generalize the symbolic heap to support partial data structures. We present the formalism as follows and illustrate how the natural proof tactics can successfully verify our running example.

3.2 Natural Proofs for Partial Data Structures and Loops.

In this paper we adopt the same proof tactics but extend symbolic heaps defined in [Madhusudan et al. 2012] to support partial data structures (e.g., list segments) and loops.

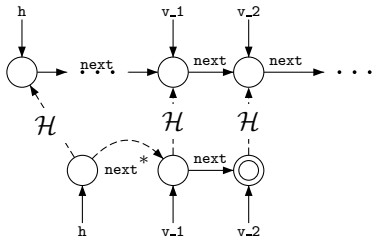
Definition 3.2 (Symbolic Heap). A symbolic heap over a set of pointer-fields Dir, a set of data-fields DF, and a set of program variables PV is a tuple (C, S, P, I, pf, df, pv) where:

- C is a finite set of *concrete nodes*, including $c_{nil} \in C$ as a special concrete node representing *nil*;;
- S is a finite set of *symbolic tree nodes* with $C \cap S = \emptyset$;
- P is a finite set of *semi-symbolic nodes* with $P \cap (C \cup S) = \emptyset$;
- $I \subseteq PV$ is a set of integer variables;
- $pf : (P \cup C \setminus \{c_{nil}\}) \times Dir \rightharpoonup C \cup S \cup P$ is a partial function mapping every pair of a concrete or semi-symbolic node and a direction to any kind of nodes;
- $df : (C \setminus \{c_{nil}\}) \times DF \to I$ is a partial function mapping concrete nodes and data-fields pairs to integer variables;
- $pv : PV \rightharpoonup C \cup S \cup P \cup I$ is a partial function mapping program variables to nodes or integer variables (location variables are mapped to $C \cup S \cup P$ and integer variables to I). □

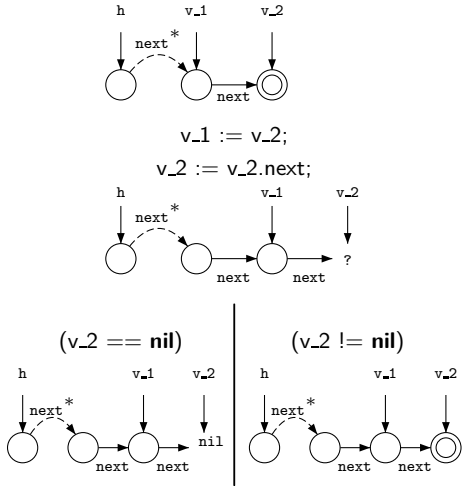
The symbolic heap has a set of concrete nodes C, a set of symbolic tree nodes S, and a set of semi-symbolic nodes P. Intuitively, a concrete node stands for a location in the concrete heap whose local fields are all known; a symbolic node stands for an arbitrary Dir-tree under it. A Semi-symbolic node p stands for the root of a partial data structure: p's concrete pointer and data fields are unknown. Moreover, if $pf(p, dir) = q$, there is an acyclic path from p to q in which there is no shared locations, which we call a *symbolic edge*. Furthermore, any symbolic tree nodes or symbolic edge represents a disjoint portion of the heap and would not intersect with each other, nor with any concrete node in C. The tree under a symbolic node or the path under a symbolic edge is not represented in the symbolic heap at all. Now assuming Dir, DF and PV are all fixed, we can define the notion of *correspondence* between symbolic heap and concrete heap:

Definition 3.3 (Heap Correspondence). Let $SH = (C, S, P, I, pf, df, pv)$ be a symbolic heap and let $CH = (N, L, pf', df', pv')$ be a concrete heap. Then CH is said to *correspond* to SH if there is a function $\mathcal{H} : C \cup S \cup P \to L$ such that the following conditions hold:

- for any $n, n' \in C \cup S \cup P$, $\mathcal{H}(n) \neq \mathcal{H}(n')$ if $n \neq n'$;
- $\mathcal{H}(c_{nil}) = nil$;



(a) Concrete Heap (above) vs. Symbolic Heap (below)



(b) Nondeterministic symbolic execution

Fig. 5. Natural Proofs for the loop invariant in srtl_insesrt_iter

- for any $n \in C \setminus \{c_{nil}\}$, and for any $dir \in Dir$, if $pf(n, dir)$ is defined, then $\mathcal{H}(pf(n, dir)) = pf'(\mathcal{H}(n), dir))$;
- similarly, for any $n \in C \setminus \{c_{nil}\}$ and any $f \in DF$, $df(n, f) = df'(\mathcal{H}(n), f)$;
- for any location variable $v \in PV$, if $pv(v)$ is defined, then $pv'(v) = h(pv(v))$;
- for any $s \in S$, $\mathcal{H}(s)$ is the root of a Dir-tree in CH (excluding c_{nil}), and any node not in the tree cannot reach the tree without reaching $\mathcal{H}(s)$;
- for any $p \in P$, and for any $dir \in Dir$, there is an acyclic path from $\mathcal{H}(p)$ to $\mathcal{H}(pf(p, dir))$, and any node not in the path cannot reach $\mathcal{H}(pf(p, dir))$ without reaching $\mathcal{H}(p)$. □

Intuitively, $\mathcal{H}$ defines a restricted kind of homomorphism between the nodes of the symbolic heap SH and the active portion of the concrete heap CH, and preserves all field access and the special nil node. The homomorphism is injective: distinct non-nil nodes are required to map to distinct locations in the concrete heap.

A key property of the correspondence between symbolic heap and concrete heap is that: when a symbolic heap is mapped to a concrete heap, the treeness and the tree-disjointness are always preserved, not only for symbolic nodes, but also for concrete nodes.

The main difference from [Madhusudan et al. 2012] 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.

3.3 An Example of Natural Proofs

Now let us graphically show symbolic heaps and natural proofs through an example srtl_insert_iter. 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 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 5a: 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 5a). The three nodes pointed by h, v_1 and v_2 are semi-symbolic node, concrete node and symbolic node, respectively. The solid edges and dashed edges represent concrete pointer fields and symbolic edges, respectively. The whole symbolic heap can be viewed as an abstraction, representing arbitrarily large concrete heaps using bounded-size graphs, and is amenable to conducting symbolic execution. There exists a correspondence $\mathcal{H}$ between the symbolic heap and the concrete heap, which is shown in Figure 5a as well. Intuitively, $\mathcal{H}$ is a homomorphism mapping each node of the symbolic heap to a node in the concrete heap. The symbolic edge labeled $\texttt{next}^*$ 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 5b. 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 $len^*(\text{v\_2})$ *always* decreases till 0. This finite model checking implies that the loop invariant is preserved on any of the resulting symbolic heaps.

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 an 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 Imp programs based on symbolic heap. The abstract semantics consists of two parts: a symbolic execution of Imp programs w.r.t. symbolic heaps, and an interpreter of Dryad$^{\text{FO}}$ formulas on symbolic heaps. Let us first formulate the ImpSynt synthesis task, then present the symbolic execution of Imp and the symbolic Dryad$^{\text{FO}}$ interpreter, and formally build the reduction and show it is sound.

4.1 The ImpSynt Synthesis Problem

The semantics of statements in Imp are 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. For example, if free(u) is executed twice in a program, the second freeing statement will always attempt to free a location already deallocated by the first statement. In that case, a runtime error is raised and the program aborts without any correctness guarantee. Note that the program contracts may use old to refer to the heap at the function entry. For example, in our running example, the postcondition contains literals like $len^*(\text{ret}) = \text{old}(len^*(\text{h})) + 1$, which can be interpreted as: evaluate $len^*(\text{ret})$ on the current heap, and evaluate $len^*(\text{h})$ on the initial heap at function entry, and the two integer values are equal.

The concrete small-step operational semantics of Imp is presented in Figure 6. We assume *ret* is a special variable for the return value. Note that we skip the rules for function calls, which requires extra bookkeeping. One can assume the standard call-by-value semantics similar to C. Then the validity of a Imp program can be formally defined as follows:

$$\frac{var' \in \mathrm{Dom}(pv)}{\langle T\ var := var', (N, L, pf, df, pv)\rangle \to \big(N, L, pf, df, pv[var \leftarrow pv(var')]\big)}$$

$$\frac{\cdot}{\langle \mathbf{loc}\ u := \mathbf{nil}, (N, L, pf, df, pv)\rangle \to \big(N, L, pf, df, pv[u \leftarrow nil]\big)}$$

$$\frac{pv(v) \in L}{\langle \mathbf{loc}\ u := v.dir, (N, L, pf, df, pv)\rangle \to \big(N, L, pf, df, pv[u \leftarrow pf(pv(v), dir)]\big)}$$

$$\frac{pv(v) \in L}{\langle \mathbf{int}\ j := v.f, (N, L, pf, df, pv)\rangle \to \big(N, L, pf, df, pv[j \leftarrow pf(pv(v), f)]\big)}$$

$$\frac{n \in N \setminus L}{\langle \mathbf{loc}\ u := \mathbf{malloc}(), (N, L, pf, df, pv)\rangle \to \big(N, L \uplus \{n\}, pf[(pv(u), f) \leftarrow nil], df[(pv(u), f) \leftarrow 0], pv[u \leftarrow n]\big)}$$

$$\frac{pv(u) \in L}{\langle \mathbf{loc}\ u.dir := v, (N, L, pf, df, pv)\rangle \to \big(N, L, pf[(pv(u), dir) \leftarrow v], df, pv\big)}$$

$$\frac{pv(u) \in L}{\langle \mathbf{int}\ u.f := v, (N, L, pf, df, pv)\rangle \to \big(N, L, pf[(pv(u), f) \leftarrow v], df, pv\big)}$$

$$\frac{l = pv(u), \quad pv(u) \in L}{\langle \mathbf{free}(u), (N, L, pf, df, pv)\rangle \to \big(N, L \setminus \{l\}, pf[\mathrm{drop}\ l], df[\mathrm{drop}\ l], pv[\mathrm{drop}\ u]\big)}$$

$$\frac{var \in \mathrm{Dom}(pv)}{\langle \mathbf{return}\ var, (N, L, pf, df, pv)\rangle \to \big(N, L, pf, df, pv[ret \leftarrow pv(var)]\big)}$$

$$\frac{\langle st\ ,\ CH\rangle \to CH'}{\langle st; blk\ ,\ CH\rangle \to \langle\ blk\ ,\ CH'\rangle}$$

$$\frac{Val(cnd) = \mathbf{true}}{\langle \mathbf{if}\ (cnd)\ \{\ blk\ \}\ \mathbf{else}\ \{\ blk'\ \}, CH\rangle \to \langle\ blk\ , CH\rangle}$$

$$\frac{Val(cnd) = \mathbf{false}}{\langle \mathbf{if}\ (cnd)\ \{\ blk\ \}\ \mathbf{else}\ \{\ blk'\ \}, CH\rangle \to \langle\ blk'\ , CH\rangle}$$

$$\frac{Val(cnd) = \mathbf{true}}{\langle \mathbf{while}\ (cnd)\ \{\ blk\ \}, CH\rangle \to \langle\ blk\ ;\ \mathbf{while}\ (cnd)\ \{\ blk\ \}, CH\ \rangle}$$

$$\frac{Val(cnd) = \mathbf{false}}{\langle \mathbf{while}\ (cnd)\ \{\ blk\ \}, CH\rangle \to CH}$$

Fig. 6. Small-step semantics of Imp .

Definition 4.1 (Program Validity). An Imp program is *valid* if for every function defined in it, if the function is called at a program state that meets the precondition (the **requires** clause), the execution will terminate at a program state that meets the postcondition (the **ensures** clause).

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 5b. Conceptually, it summarizes all possible executions on all concrete heaps corresponding to the symbolic heap. As we have extended the symbolic heap presented in [Madhusudan et al. 2012] with symbolic edges, we also need to extend symbolic execution to handle not only symbolic nodes but also symbolic edges. Recall that a symbolic heap *SH* consists of concrete, symbolic and semi-symbolic nodes. Concrete nodes correspond to concrete locations and their pointer and data fields are all determined; while symbolic and semi-symbolic nodes correspond to arbitrary full trees or partial trees, hence the pointer and data fields are unknown. Therefore, if a statement accesses the next field of a concrete node, this manipulation can simply mimic the semantics on concrete heaps; if it manipulates on a symbolic node or a symbolic edge, we need to unfold this node or edge. For example, in Figure 5b, 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 5b.[3]

Figure 7 illustrates the nondeterministic symbolic execution of Imp with respect to symbolic heaps. Note that for the dereference statement $u := v.dir$ or $j := v.f$, if v is a symbolic node, there is a preprocessing step to unfold v nondeterministically, considering both the nil and non-nil cases for every pointer field. For other statements that involve field accessing, to simplify the symbolic execution and its encoding in the following section, we assumes that the involved node v is always concrete. This assumption is reasonable as we can always instrument a statement of the form $temp := v.dir$ right before the current statement, where *temp* is a temporary variable, to make sure v is already concretized. As the field accessing statement has no side effect, the semantics of the program is not affected.

Symbolic function calls are also nondeterministic: the portion of the heap that can be touched by the function call will be havoc-ed; and the new heap after the call must satisfy the postcondition of the call, w.r.t. an oracle function, which will be explained later in this section.

The nondeterministic symbolic execution is not defined for while-loops, i.e., we can simulate the real execution of any loop-free block in Imp . 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 is verifiable that the nondeterministic function call overapproximates all possible real function calls. In other words, with function calls, the soundness of the symbolic execution still holds.

It can be verified that the symbolic execution mimics real executions on concrete heaps precisely. More formally, the following lemma can be proved:

Lemma 4.2. *Given a symbolic heap SH and an* Imp *basic block B without any function call, the correspondence between SH and its concrete heaps is maintained after executing B. In other words, for any concrete heap CH, it can be obtained by executing B starting from a concrete heap corresponding to SH, if and only if the symbolic execution of B starting from SH may yield a symbolic heap to which CH corresponds to.*

[3] Another source of nondeterminism is the value of h.key, which is not reflected in Figure 5b.

$$\frac{var' \in \mathrm{Dom}(pv)}{\langle var := var', \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf, df, pv[var \leftarrow pv(var')]\big)}$$

$$\frac{\cdot}{\langle \mathbf{loc}\ u := \mathbf{nil}, \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf, df, pv[u \leftarrow nil]\big)}$$

$$\frac{pv(v) \in S \cup P, \quad \text{there is } n_{dir} \notin C \cup S \cup P, \text{for each } dir \in Dir}{\begin{array}{c}\langle \mathbf{loc}\ u := v.dir,\ \big(C, S, P, I, pf, df, pv\big)\rangle \\ \to \langle \mathbf{loc}\ u := v.dir,\ \big(C \uplus \{n_{dir} \mid dir \in Dir\}, S \setminus \{pv(v)\}, P \setminus \{pv(v)\}, I, pf, df, pv\big)\rangle\end{array}}$$

$$\frac{pv(v) \in C \setminus \{c_{nil}\}}{\langle \mathbf{loc}\ u := v.dir,\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf, df, pv[u \leftarrow pf(pv(v), dir)]\big)}$$

$$\frac{pv(v) \in S \cup P, \quad \text{there is } n_{dir} \notin C \cup S \cup P, \text{for each } dir \in Dir}{\begin{array}{c}\langle \mathbf{int}\ u := v.f,\ \big(C, S, P, I, pf, df, pv\big)\rangle \\ \to \langle \mathbf{int}\ u := v.f,\ \big(C \uplus \{n_{dir} \mid dir \in Dir\}, S \setminus \{pv(v)\}, P \setminus \{pv(v)\}, I, pf, df, pv\big)\rangle\end{array}}$$

$$\frac{pv(v) \in C}{\langle \mathbf{int}\ u := v.f,\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf, df, pv[u \leftarrow pf(pv(v), f)]\big)}$$

$$\frac{n \notin C \cup S \cup P}{\begin{array}{c}\langle u := \mathbf{malloc}(),\ \big(C, S, P, I, pf, df, pv\big)\rangle \\ \to \big(C \uplus \{n\}, S, P, I, pf[(pv(u), f) \leftarrow nil], df[(pv(u), f) \leftarrow 0], pv[u \leftarrow n]\big)\end{array}}$$

$$\frac{pv(u) \in C \setminus \{c_{nil}\}}{\langle u.dir := v,\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf[(pv(u), dir) \leftarrow v], df, pv\big)}$$

$$\frac{pv(u) \in C}{\langle u.f := j,\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf[(pv(u), f) \leftarrow j], df, pv\big)}$$

$$\frac{\text{there is } pv(u) = l, \quad pv(u) \in C}{\langle \mathbf{free}(u),\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C \setminus \{l\}, S, P, I, pf[\mathrm{drop}\ l], df[\mathrm{drop}\ l], pv\big)}$$

$$\frac{var \in \mathrm{Dom}(pv)}{\langle \mathbf{return}\ var,\ \big(C, S, P, I, pf, df, pv\big)\rangle \to \big(C, S, P, I, pf, df, pv[ret \leftarrow pv(var)]\big)}$$

$$\frac{\langle st\ ,\ SH\rangle \to SH'}{\langle st; blk\ ,\ SH\rangle \to \langle\ blk\ ,\ SH'\rangle}$$

$$\frac{Val(cnd) = true}{\langle \mathbf{if}\ (cnd)\ \{\ blk\ \}\ \mathbf{else}\ \{\ blk'\ \}, SH\rangle \to \langle\ blk\ , SH\rangle}$$

$$\frac{Val(cnd) = false}{\langle \mathbf{if}\ (cnd)\ \{\ blk\ \}\ \mathbf{else}\ \{\ blk'\ \}, SH\rangle \to \langle\ blk'\ , SH\rangle}$$

Fig. 7. Symbolic execution of Imp .

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* as a symbolic node (edge) may represent an arbitrary tree (list segment); therefore the values of recursive definitions on it cannot be determined. For example, if n is a symbolic node representing a list, then $max^*(n)$ can return any integer, as any integer can be the max value of a list.

In that case, we assume the oracle function we mentioned previously can interpret it correctly in any context. For the $max^*(n)$ example, we assume the max value of n is captured by $Orcl(max^*, n)$. 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. Intuitively, The partial interpretation 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.

The details of the algorithm are shown in Algorithm 1. Let us start from interpreting recursive definitions, which is the most interesting case. We define a function `interpret`$(SH, Orcl, R^*, n)$ that partially interprets a recursive predicate R^* on a node n in a symbolic heap SH via inquiries to the oracle function *Orcl*.[4]

Remember that evaluating $R^*(n)$ requires that n subtends a tree. Thus the interpretation starts by asserting the treeness of the reachable nodes from n, which is the necessary condition for interpreting $R^*(n)$ according to DRYAD^{FO} semantics. The treeness is checked by an auxiliary function `checkTreeness`, which is basically a procedure for computing the transitive closure starting from n. It is clear that the treeness holds on the symbolic heap if and only if every node is visited up to once. If the treeness holds, the interpretation proceeds with a case analysis on the type of n. If n is a concrete node, the recursive definition can be recursively computed: if n is *nil*, follow the base case, otherwise unfold the recursive definition and interpret recursively on n's neighbors. Finally, when n is a symbolic node or a semi-symbolic node, simply inquire of the *Orcl* function.

The `interpret` function can be easily extended to a partial interpreter for all DRYAD^{FO} formulas, as the other constructs of the logic are just standard integer and boolean operands, and hence can be interpreted straightforwardly.

4.4 The Natural Synthesis Problem

Now to define the natural synthesis problem, we first introduce *normal formulas* and *normal programs*: 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. Note that for a normal program, its program states can be characterized using a bounded number of symbolic heaps. We also introduce the notion of *symbolic validity*:

Definition 4.3 (Symbolic Validity). An IMP program is *symbolically valid* if for every basic block of the program, for every nondeterministic symbolic execution of the basic block and every *Orcl* function that returns valid values, if the execution starts from an symbolic heap satisfying the precondition, then the execution will end with a symbolic heap satisfying the postcondition.

Then the natural synthesis problem is to synthesize a *normal and symbolically valid* IMP program.

[4]To simplify the presentation, we consider only unary predicates in this section; but our results can be extended to other recursive definitions easily.

input : A symbolic heap $SH = (C, S, P, I, pf, df, pv)$,
An oracle function $Orcl$ answering any inquiry of the form $Orcl(T^*, nn)$,
A DRYAD^{FO} recursive definition R^*,
A node $n \in C \cup S \cup P$
output : $R^*(n)$

```
def interpret(SH, Orcl, R*, n):
    assert checkTreeness(SH, n)
    if n = c_nil :
        return R_base(n)
    elif n ∈ C ∪ P :
        φ ← R_ind(n)
        for T*(x.dir) occurred in R_ind(x) :
            neighbor ← pf(n, dir)
            value ←
              interpret(SH, Orcl, T*, neighbor)
            φ ← φ[T*(n)/value]
        for x.f occurred in R_ind(x) :
            intval ← df(n, f)
            φ ← φ[n.f/intval]
        return the value of φ in FOL
    else:
        return Orcl(R*, n)
```

```
def checkTreeness(SH, n):
    reach ← [false] * |C ∪ S ∪ P|
    reach(n) ← true
    flag ← true
    while flag :
        flag ← false
        for c ∈ C ∪ P, dir ∈ Dir :
            if pf(c, dir) ∈ C ∪ P \ {c_nil} :
                if reach[pf(c, dir)] :
                    return false
                else:
                    reach[pf(c, dir)], flag ← true
    return true
```

Algorithm 1: Partial interpretation of recursive definitions on symbolic heaps

THEOREM 4.4. *The reduction from an* IMPSYNT *synthesis problem to its corresponding natural synthesis problem is sound.* □

PROOF. We need to show that 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. By contradiction, assume S is not valid, then there exists an execution of a function which violates the ensures claim. In addition, this execution must have witnessed the invalidity of a Hoare-triple. As S is normal, the initial heap satisfies the Hoare-triple's precondition and can be summarized by a symbolic heap. Then by Lemma 4.2, this execution has been summarized by one of the non-deterministic symbolic executions, and the *Orcl* function can simply borrow valid values from the real execution. By Definition 4.3, S violates the symbolic validity. The contradiction concludes the proof. □

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. 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 [Solar-Lezama et al. 2006] or ROSETTE [Torlak and Bodik 2014]. 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.

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.[5] We represent these bounds as B_{HEAP}, B_{LVAR}, B_{IVAR}, B_{DIR} and B_{DATA}, respectively, and declare the following arrays:

int[B_{LVAR}] *locvars*; **int**[B_{IVAR}] *intvars*;
bit[B_{HEAP}] *active*;
bit[B_{HEAP}] *symbolic*; **bit**[B_{HEAP}] *semisym*;

Intuitively, every location/integer variable is assigned an id less than B_{LVAR}/B_{IVAR}, and every heap location is assigned an id less than B_{HEAP}. Then *locvars* / *intvars* represents a variable store that maps every variable id to its corresponding value. The two bit vectors *active*, *symbolic* and *semisym* characterize the $C \cup S \cup P$ set, the S set and the P set in the current symbolic heap, respectively. Note that when $active[n]$ is false, the location is not active in the current heap, and hence $symbolic[n]$ can be arbitrary. In particular, let the special concrete node c_{nil} be always active and has a special id 0 . Let us also assume there is a special variable *nil* with id 0 and always maps to c_{nil}:

assume $locvars[0] = 0 \wedge active[0] \wedge \neg symbolic[0] \wedge \neg semisym[0]$;

Moreover, pointer fields and data fields can be represented using two separate two-dimension arrays:

int[B_{HEAP}][B_{DIR}] *dirs*;
int[B_{HEAP}][B_{DATA}] *data*;

dirs stores location id's pointed to by each pointer field of each heap location. *data* stores simply integer values for each location's each data field. We should assure that the id's stored in *dirs* are less than B_{HEAP}:

assume $\bigwedge_{i<B_{HEAP},\ j<B_{DIR}} 0 \leq dirs[i][j] \leq B_{HEAP}$;

Notice that there is no constraint for all symbolic nodes and the special concrete node c_{nil}, as all their fields are meaningless and can be *arbitrary*.

Encoding Symbolic Semantics. We can simulate the symbolic execution presented in Section 4.3 as manipulations on the arrays. 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.

Other statements can be simulated using separate functions similarly. The only noteworthy case is $u = v.dir$. As we have shown in Section 4.2, its symbolic execution is nondeterministic. When v is a symbolic node, it should be unfolded first, then each pointer field of $v.dir$ can either be c_{nil} or a fresh symbolic node, and each data field can be an arbitrary integer. We use two functions is_nil(v, *dir*) and get_int(v, *df*) to explore the above nondeterministic cases thoroughly. Then the simulation looks as follows:

```
void loc_deref(int u, int v, int dir) {
  ... // implement u := v.dir
  int source = locvars[u];
  ...
```

[5] 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.

```
    if (symbolic[source]) {
      symbolic[source] = false;
      for (int 0 ≤ pfld < B_DIR) { // for each pointer field
        if (is_nil(source, pfld))  dirs[source][pfld] = 0;
        else {
          int fresh = ...; // find an inactive loc
          active[fresh] = true;
          symbolic[fresh] = true;
          dirs[source][pfld] = fresh;
        }
      }
      for (int 0 ≤ dfld < B_DATA) { // for each data field
        dirs[source][dfld] = get_int(source, dfld);
      }
  ... }
    else { ... }
}
```

Encoding Function Calls and* DRYAD^{FO} *Logic. For function calls, we create a special havoc function to nondeterministically (will explain shortly) update all reachable locations from the call arguments. Then presuming that assumptions and assertions are supported in the synthesis engine, 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). For example, the effect of calling srtl_insert_rec(h, t) can be simulated by:

```
void call_srtl_insert_rec(int h, int t, int ret,
  ref int[LVAR] old_locvars, ref int[IVAR] old_intvars,
  ref int[HEAP] old_symbolic, ref int[HEAP] old_semisym)
{
  assert interpret_sorted_l(h, locvars, intvars, ...);
  snapshot(old_locvars, old_intvars, old_symbolic, old_semisym);
  locvars[ret] := havoc(h, old_locvars, old_intvars, old_symbolic, old_semisym);
  assume interpret_sorted_l(ret, locvars, ...)
      ∧ interpret_len(ret, locvars, ...) = interpret_len(h, old_locvars, ...) + 1 ∧ ...;
}
```

where interpret_sorted_l($h, \dots$) interprets $sorted\_l^*(h)$ on the symbolic heap described by the rest of the arguments. The implementation essentially encodes the symbolic interpreter described in Section 4.3 modulo the *Orcl* function. For every inquiry $Orcl(sorted\_l, n)$ to the oracle for a symbolic node n, interpret_sorted_l replace it with a special function sym_sorted_l(n), which guarantees to explore all possible values might be returned by the inquiry. Other parts of the interpreter can be easily encoded–all the integer and boolean connectives are native operations in FOL. Specifically, the term nil can be interpreted as 0. For field dereference, take $v.dir$ as an example, then the field accessing is simply $dirs[locvars[v]][dir]$.

Uninterpreted Functions. The natural synthesis problem models nondeterministic choices using uninterpreted functions, which can be interpreted *arbitrarily*. For example, the special function *sym_sorted_l*(n) we mentioned above is uninterpreted and may return either true or false. Nondeterminism is also involved in the symbolic execution, such as integer field dereference, so that all possible nondeterministic executions can 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.

***Encoding Unknown Holes in* ImpSynt.** Recall that our goal is to encode an ImpSynt synthesis problem. We have shown how to encode statements in Sketch and formulas specified in Dryad^{FO}; now what remain are the unknown holes allowed in ImpSynt , including ??, **cond**(), **val**(), **stmt**(C), **conj**(C), and **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. Then a hole **cond**() can be translated to a call to the following generator function with a predefined constant bound *cb* (assuming variable equality check only, written in Sketch syntax):

```
generator bit cond() {
  for (int 0 ≤ i < cb) {
    if (gen_equality()) return false;
  }
  return true;
}
generator bit gen_equality() {
  int bound = {B_LVAR | B_IVAR};
  bit eq = gen_var(bound) == get_var(bound);
  if (??) return eq else return ¬eq;
}
generator int gen_var(int bound) {
  int result = ??;
  assert 0 ≤ result < bound;
  return result;
}
```

The generator function picks up to *cb* number of equality checks between location variables or integer variables, chooses to negate some of them, and checks if they are all true. Note that the generator can easily pick a tautology $u = u$ to reduce the number of conjuncts, or pick a contradiction $u \neq u$ to get a false condition.

The conjunction hole **conj**(C) can be generated similarly: simply replace gen_equality() with a similar function, say gen_literal(), which may generate not only variable equalities but also recursive definitions on arbitrary location variable. There should be assertions as well confirming the generated formulas are always normal. For the interest of space, the details are omitted. The holes in the ranking function can be generated in the same fashion. The space of **stmt**(C) can also be described similarly.

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 solution to the natural synthesis problem, and thus also a valid solution for the original ImpSynt synthesis problem (by Theorem 4.4).

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. *From left to right:* If sk has a normal and symbolically valid solution, the solution can be mapped to a particular unknown assignment to *encode*(sk), resulting in a finite circuit p. Consider evaluating p with arbitrary inputs and uninterpreted functions. It can be verified that it actually simulates a symbolic execution of Imp programs with a particular *Orcl* function. Therefore, when the filled sk program is symbolically valid, by definition, this particular symbolic execution does not violate any assertion. As all assertions on symbolic heaps have been equivalently encoded to *encode*(sk), there is no assertion violated and p will always be evaluated `true`, i.e., p is a solution to *encode*(sk).

From right to left: If *encode*(sk) has a solution, it reflects a solution to the original sk program, which we claim is normal and symbolically valid. Notice that *encode*(sk) guarantees the filled program is normal, i.e., all preconditions of the filled sk can be translated to a bounded number of initial symbolic heaps, and of bounded sizes and bounded number of variables. Moreover, consider arbitrary symbolic execution E of arbitrary basic block, as the length of the basic block is finite, every intermediate symbolic heap is of bounded size and of bounded variables. In other words, if the bounds in our encoding are chosen large enough, E can always be simulated in *encode*(sk), which does not violate any assertion. Neither does E. □

6 EXPERIMENTS

We pick Sketch [Solar-Lezama et al. 2006], a state-of-the-art inductive synthesizer, as the back-end synthesis engine of our natural 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 [Solar-Lezama et al. 2006] 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 is 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 [Solar-Lezama et al. 2006], 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, Theorems 5.1 and 4.4 imply that we can algorithmically find normal and symbolically valid solutions for ImpSynt sketches, which 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.

Malloc Budget. Ideally, the heap size bound $\mathsf{B}_{\mathsf{DIR}}$ is determined by the largest possible size of a basic block, as each statement can modify a fresh new node. This bound can completely avoid the out-of-memory error but is usually too large to be handled in practice. To keep the heap size bound small yet avoid the out-of-memory error, we introduce a "malloc budget" as the number of unassigned locations. A malloc operation or unfolding a symbolic node succeeds only if the malloc budget is nonempty; otherwise we regard the symbolic execution as failed and the program as not satisfying the specification. For example, an instruction **loc** u := **malloc**() can be simulated as follows:

```
void malloc(int u) {
  if (malloc_budget < 1) return; // check the malloc budget
  malloc_budget−−;

  int fresh = ...; // find an inactive loc
  active[fresh] = true;
  symbolic[fresh] = false;
  for (int 0 ≤ pfld < B_DIR) dirs[source][pfld] = 0; // initialize each pointer field
}
```

Then $\mathsf{B}_{\mathsf{DIR}}$ can be improved to $n + mb$, where n is the number of location variables involved in a basic block and mb is the malloc budget. We observed that in most programs we synthesized the malloc budget just needs to be 1. Hence our synthesizer always try the malloc budget 1 first. If the number is too small and the synthesizer can't find a solution, it increases the number and retry, until a solution is found.

Axioms and Assumptions. 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. In addition, for each old(V) term appeared in the postcondition, we may create a snapshot old_V at function entry, and assume the value is always remembered by an unknown term T through a loop. In other words, there must be a conjunct old_V = T in the loop invariant.

Ranking Functions. We also observed that the search space for the ranking function can dramatically affect the performance. Hence we assume a template for the ranking function: $f^*(v)$, where f^* is a recursive function and v is a mutable variable involved in the loop. We believe this is a reasonable assumption: one the one hand, the ranking function has to involve a variable that is mutated in the loop body so that its value gets reduced in each loop iteration; on the other hand, the ranking function is usually a nonnegative measurement such as the length or size for a portion of the heap. For instance, in the example srtl_insert_iter, the loop body mutates v_2 and the ranking function is just the length of the list: len^*(v_2). This assumption works for all programs we synthesized.

		ImpSynt					Sketch			
	Example	full spec?	#call	#cond	#loop	#stmt	hole size	#iter	time	authentic solution?
List	sll_max_rec	Y	1	2	0	3	49	13	11s	Y
	sll_max_iter	Y	0	0	1	0	39	7	27s	Y
	sll_min_rec	Y	1	2	0	3	48	22	23s	Y
	sll_min_iter	Y	0	0	1	0	41	13	33s	Y
	sll_len_rec	Y	1	1	0	3	44	11	12s	Y
	sll_len_iter	Y	0	0	1	0	29	18	26s	Y
Sorted List	srtl_prepend	N	0	0	0	0	4	3	8s	Y
	srtl_reverse_iter	N	0	0	1	0	90	71	15m56s	Y
	srtl_insert_rec	N	1	1	0	2	44	12	28s	Y
	srtl_insert_iter	N	0	1	1	1	74	21	63m24s	Y
	srtl_merge_rec	N	1	2	0	8	112	-	Timeout	-
	insertion_sort_rec	N	2	1	0	2	44	16	1m34s	Y
BST	bst_left_rotate	N	0	0	0	3	25	3	17s	Y
	bst_right_rotate	N	0	0	0	3	25	4	14s	Y
	bst_insert_rec	N	1	0	0	5	72	17	5m43s	Y
	bst_del_root_rec	N	1	0	0	3	53	32	40m16s	Y

Fig. 8. List of synthesized provably-correct data-structure manipulations. (Program templates and generated Sketch files are available as supplemental material for this paper)

6.2 Experimental Evaluation

To demonstrate the effectiveness of our natural synthesis approach and our synthesizer, we conducted experiments on synthesizing 16 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 its 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 write even 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 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 synthesizes sorted list manipulations such as prepend, reverse, insert, or insertion_sort. 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_iter takes 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_rec is expected to implement the insertion sort algorithm in a recursive manner. It is noteworthy that insertion_sort_rec is synthesized modularly: the template does insertion by calling srtl_insert, whose implementation is synthesized separately. 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, including left-rotate, right-rotate, recursive key insertion, and root deletion. 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) child as the new root. The example bst_insert_rec recursively inserts 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. Note that for the last two sets of examples, our specifications are rich but incomplete, as Dryad^{FO} is unable to specify the elements of the output list/tree, which requires a theory of sets.

The experiments were conducted on a 10-core, 96GB server running Ubuntu 14.04.5 and Sketch 1.7.2. We summarize our experimental results in Figure 8. For each example, the ImpSynt part reports some numbers about the specification. The first column indicates whether the template expresses full functional properties; the next column indicates the number of function calls with unknown parameters, e.g., the call srtl_insert_rec(??, ??) in Figure 3. The next three columns show number of unknown conditionals (**cond**()), unknown loops (**loop**(V, N) or **simple–loop**(V, N)) and unknown statements (**stmt**(1)), respectively. Note that a sequence of unknown statement **stmt**(C) can be decomposed to C unknown statements in a row. The numbers V, N, C are all reasonably estimated based on our programming experiece. For example, to synthesize a tree rotation routine, we believe it should consist of up to 3 statements. The Sketch part reports some numbers about solving the synthesis problem. The hole size column reports the number of unknown bits of the internal model in Sketch, which represents the search space of the synthesis problem. For example, encoding sll_max_rec yields 49 unknown bits, i.e., there are 2^{49} candidate programs. The next two columns show the number of iterations performed by the CEGIS engine and the total time spent by Sketch (time for encoding/decoding is negligible) with one hour timeout. We only report the time spent by Sketch because the time spent by other phases of the system is negligible. The last column shows whether the synthesized program is the authentic implementation that one can expect.

The synthesis results are encouraging: our system successfully produced verified implementations as well as necessary loop invariants and ranking functions for all but one program sketches. 10 out of 14 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. Despite of incomplete specifications for some examples, all the synthesized implementations are authentic 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 guaranteed.

6.3 Limitations

One limitation of the system is the expressivity of the logic Dryad^{FO}. As a first-order logic, it is not capable of describing the full specification of sorted list and BST examples, i.e., describing the sets/multisets of the contents stored in the list or tree. In its stead, we use max^* and min^* to characterize the expected set/multiset, which is unnatural and imprecise. In the future we plan to extend our system to support the full Dryad logic and set/multiset specifications.

Towards synthesizing larger and more realistic programs, the biggest limiting factor is the search space – the synthesis engine will time out with more sophisticated data structures and synthesis tasks. That said, our system may scale better in the context of "repairing" a buggy program. In other words, if the aim is to fix a buggy program, one can represent all possible patches as a template and

find the correct patch using the ImpSynt synthesizer. Then the synthesizer may keep the search space tractable and generate larger programs. We plan to explore this direction in the future.

Another limiting factor is the unsupported program structures. For example, our system does not support nested loops, which is necessary for almost every sorting algorithm. This is another direction we will explore in the future.

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* follows the correct-by-construction methodology and derives a correct program from a set of inference rules. In contrast, *Inductive synthesis* separates program synthesis and verification explicitly, by an "enumerate-and-verify" loop. 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. [2013] 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. Synquid [Polikarpova et al. 2016] adopts similar synthesis strategy and generates more sophisticated recursive functional programs such as insertion to sorted lists and BSTs, and guarantees their correctness. 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, not all program synthesized by their system are automatically verified; in contrast, our system always guarantee the synthesized programs are provably correct, and no additional verification required. $\lambda 2$ [Feser et al. 2015] is another synthesizer that produces functional programs over recursive data-structures from examples.

A more recent work by Delaware Delaware et al. [2015] 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 [Hawkins et al. 2011] 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* [Srivastava et al. 2010]. 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 [Singh and Solar-Lezama 2011; Vechev and Yahav 2008; Vechev et al. 2007], 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 [Singh and Solar-Lezama 2011] 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 [Lev-Ami et al. 2000; Lev-Ami and Sagiv 2000], 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 [Madhusudan et al. 2012; Qiu et al. 2013] which have been elaborated in Section 3. A standalone natural-proof verifier [Pek et al. 2014] that requires only pre-/post-conditions and loop invariants was also built based on VCC [Cohen et al. 2009]. Similar proof tactics by unfolding recursive definitions were also explored in [Chin et al. 2012; Suter et al. 2011]. Other program verifiers [Chlipala 2011; Jacobs et al. 2011] based on separation logic [O'Hearn et al. 2001; Reynolds 2002] 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 [Iosif et al. 2013] and [Madhusudan et al. 2011] are very powerful.

8 CONCLUSION

We present Natural Synthesis, a technique that brings natural proofs to program synthesis. We aim to synthesize programs whose correctness can be proved using natural proofs, a useful methodology for program verification. We show this problem can be encoded to SKETCH, a modern inductive synthesis framework. Our system successfully synthesized provably-correct programs manipulating sorted lists and binary search trees, either recursively or iteratively. These programs 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.

ACKNOWLEDGMENTS

We would like to thank Hong Wang, an undergraduate student at Nanjing University, for his help in packaging the supplemental material for this paper.

This material is based upon work supported by the National Science Foundation under Grant No. 1139056 (ExCAPE). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Sci. Comput. Program.* (2012), 1006–1036.

Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/1993498.1993526

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs'09 (LNCS)*, Vol. 5674. Springer, Berlin, Heidelberg, 23–42.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural Proofs for Asynchronous Programs Using Almost-synchronous Reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 709–725. https://doi.org/10.1145/2660193.2660211

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1993498.1993504

Radu Iosif, Adam Rogalewicz, and Jirí Simácek. 2013. The Tree Width of Separation Logic with Recursive Definitions. In *CADE-24*. Springer, Berlin, Heidelberg, 21–38.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*. Springer, Berlin, Heidelberg, 41–55.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. 2000. Putting Static Analysis to Work for Verification: A Case Study. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. ACM, New York, NY, USA, 26–38. https://doi.org/10.1145/347324.348031

Tal Lev-Ami and Shmuel Sagiv. 2000. TVLA: A System for Implementing Static Analyses. In *SAS '00 (LNCS)*, Vol. 1824. Springer, Berlin, Heidelberg, 280–301. https://doi.org/10.1007/978-3-540-45099-3_15

P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. 2011. Decidable Logics Combining Heap Structures and Data. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 611–622. https://doi.org/10.1145/1926385.1926455

P. Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. 2012. Recursive Proofs for Inductive Tree Data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 123–136. https://doi.org/10.1145/2103656.2103673

Zohar Manna and Richard Waldinger. 1979. Synthesis: Dreams => Programs. *IEEE Transactions on Software Engineering* 5, 4 (1979), 294–328.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL'01 (LNCS)*, Vol. 2142. Springer, Berlin, Heidelberg, 1–19.

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/2491956.2462169

John C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 313–326. https://doi.org/10.1145/1706299.1706337

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *SAS'11 (LNCS)*, Eran Yahav (Ed.), Vol. 6887. Springer, Berlin, Heidelberg, 298–315.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

Martin Vechev and Eran Yahav. 2008. Deriving Linearizable Fine-grained Concurrent Objects. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 125–135. https://doi.org/10.1145/1375581.1375598

Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. 2007. CGCExplorer: A Semi-automated Search Procedure for Provably Correct Concurrent Collectors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 456–467. https://doi.org/10.1145/1250734.1250787