Natural Synthesis of Provably-Correct Data-Structure Manipulations

Supplemental Materials

A. The DRYAD$^{\text{FO}}$ Logic

A.1 Syntactic restrictions on recursive 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.

A.2 Semantics of DRYAD$^{\text{FO}}$

Formally, a concrete heap is defined as follows ($f : A \rightharpoonup B$ denotes a partial function from A to B):

Definition A.1 (Concrete Heap)**.** *A concrete heap over a set of pointer-fields PF, 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);*
- $pf : (L \setminus \{nil\}) \times PF \to N$ *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 *PF*. 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 *PF*, it will evaluate to `undef` if *lt* evaluates to `undef`, or the subgraph reachable from lt via *PF* 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 $\texttt{disjoint}(x, y)$ 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:

$$tree^*(x) \stackrel{def}{=} \texttt{its}(x = \texttt{nil}, \texttt{true}, \texttt{true})$$

$$\mathit{tree}^*(x,y) \stackrel{def}{=} \mathtt{its}(x = y, \mathtt{true}, \mathtt{true})$$

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

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

$$\mathit{reach}^*_u(x) \stackrel{def}{=} \mathtt{ite}\big(x = \mathtt{nil}, \mathtt{false}, \mathtt{x} = \mathtt{u} \vee \bigvee_{dir \in PF} \mathit{reach}^*_u(x.dir)\big)$$

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

B. Symbolic Heap

B.1 Formal Definitions

Definition B.1 (Symbolic Heap)**.** *A symbolic heap over a set of pointer-fields PF, 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* : $(P \cup C \setminus \{c_{nil}\}) \times PF \rightharpoonup C \cup S \cup P$ *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*).* □

Intuitively, 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, with the idea that each symbolic node stands for an arbitrary *PF*-tree under it, and each semi-symbolic node reaches its *dir* field with an acyclic path in which there is no shared locations, which we call *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 *PF*, *DF* and *PV* are all fixed, we can define the notion of *correspondence* between symbolic heap and concrete heap:

Definition B.2 (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* $n \neq n'$*;*
- $\mathcal{H}(c_{nil}) = nil$*;*
- *for any* $n \in C \setminus \{c_{nil}\}$*, and for any dir* $\in$ *PF, 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 PF-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* $\in$ *PF, 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 preserved not only for symbolic nodes, but also for concrete nodes.

For a symbolic heap $SH = (C, S, P, I, pf, df, pv)$, let the set of *graph nodes* of *SH* be the smallest set of nodes $V \subseteq C \cup S \cup P$ such that:

- $S \cup \{c_{nil}\} \subseteq V$
- For any node $n \in C \cup P$, if for every *dir* $\in$ *PF*, $pf(n, dir)$ is defined and belongs to V, then $n \in V$.

Definition B.3 (Treeness Graph)**.** *Given a symbolic heap SH, its* treeness graph *TGraph*(SH) *is a directed graph* (V, E)*, where* V *is the set of graph nodes defined above, and* E *is the set of edges* (u, v) *such that* $pf(u, dir) = v$ *for some dir* $\in$ *PF.*

A more constructive way to obtain the treeness graph is to recursively remove concrete nodes that has missed pointer fields (undefined or pointed to a removed node). We say that a node u in V *subtends* a tree in *TGraph*(SH) if the set of all nodes reachable from u forms a tree (in the usual graph-theoretic sense).

The crux of natural proofs is that: the invariant preservation on the symbolic heap immediately implies the invariant preservation on *arbitrary* concrete heap. We will show the reduction more formally in next section. The proof critically relies on the fact that the homomorphism $\mathcal{H}$ preserves both *treeness* and *tree-disjointness*. The two properties can be formally described by the following two lemmas:[1]

Lemma B.4 (Treeness Preservation)**.** *Let SH be a symbolic heap and let CH be a corresponding concrete heap, with respect to a mapping function $\mathcal{H}$. If a node u subtends a tree in TGraph(SH), then $\mathcal{H}(u)$ also subtends a tree in CH.*

Lemma B.5 (Disjointness Preservation)**.** *Let SH be a symbolic heap and let CH be a corresponding concrete heap, with respect to a mapping function $\mathcal{H}$. If a node u subtends a tree in TGraph(SH), and v is another node in TGraph(SH) but not in the tree under u, $\mathcal{H}(v)$ is not in the tree under $\mathcal{H}(u)$ either. Moreover, if there are two disjoint trees under u and v in TGraph(SH), then $\mathcal{H}(u)$ and $\mathcal{H}(v)$ also subtend two disjoint trees in CH.*

Proof. We prove the above two lemmas together by induction on the height of the tree under u.

Base Case: If the height is 0 ($u = c_{nil}$) or 1 ($u \in S$), by Definition 3.2, $\mathcal{H}(u)$ is the root of a *PF*-tree, and is disjoint from any other part of *CH*.

Induction Step: Assume the treeness and disjointness are preserved for all tree of height up to k, then consider a node u in *TGraph*(*SH*), with a tree of height $k+1$ under it. By Definition B.3, all the pointer fields of u are defined, i.e., for every $dir \in PF$, $pf(n, dir)$ is the root of a subtree in *TGraph*(*SH*), with height no more than k. Due to the treeness of u and the inductive hypothesis, all these subtrees do not contain u, and are disjoint from each other. Now by Definition 3.2, every $\mathcal{H}(pf(n, dir))$ subtends a tree in *CH* as well. Moreover, the treeness of $\mathcal{H}(u)$ is also preserved: every $\mathcal{H}(pf(n, dir))$ does not contain $\mathcal{H}(u)$, and they are all disjoint from each other.

The disjointness preservation can be proved similarly. □

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

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

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

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

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

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

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

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

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

Figure 1: Small-step semantics of IMP .

C. Concrete and Symbolic Semantics of IMP

C.1 Operational Semantics

The concrete small-step operational semantics of IMP is presented in Figure 1 and 2. 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.

The validity of a IMP program can be formally defined as follows:

Definition C.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).*

C.2 Symbolic Execution

Figure 3 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

[1] *TGraph*(*SH*) is the treeness graph of *SH*, defined in Appendix B.

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

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

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

$$\frac{Val(cnd) = \text{true}}{\begin{array}{c}\langle \texttt{while}\ (cnd)\ \{\ blk\ \}, CH\rangle \\ \to \langle\ blk\ ;\ \texttt{while}\ (cnd)\ \{\ blk\ \}, CH\ \rangle\end{array}}$$

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

Figure 2: Small-step semantics of IMP (continued).

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.

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

Lemma C.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.*

C.3 Symbolic Interpretation

We partially interpret DRYAD$^{\text{FO}}$ formulas on the symbolic heap, as shown in Algorithm 1. Let us start from interpreting recursive definitions, which is the most interesting case. We define a function $\texttt{interpret}(SH, Orcl, R^*, n)$ that partially interprets a recursive predicate R^* on a node n in a symbolic heap *SH* via inquiries to an oracle function *Orcl*.[2]

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$^{\text{FO}}$ 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

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

$$\frac{var' \in \text{Dom}(pv)}{\begin{array}{c}\langle var := var', (C, S, P, I, pf, df, pv)\rangle \\ \to (C, S, P, I, pf, df, pv[var \leftarrow pv(var')])\end{array}}$$

$$\frac{\cdot}{\begin{array}{c}\langle \texttt{loc}\ u := \texttt{nil}, (C, S, P, I, pf, df, pv)\rangle \\ \to (C, S, P, I, pf, df, pv[u \leftarrow nil])\end{array}}$$

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

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

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

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

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

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

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

$$\frac{\text{there is } pv(u) = l, \quad pv(u) \in C}{\begin{array}{c}\langle \texttt{free}(u),\ (C, S, P, I, pf, df, pv)\rangle \\ \to (C \setminus \{l\}, S, P, I, pf[\text{drop } l], df[\text{drop } l], pv)\end{array}}$$

$$\frac{var \in \text{Dom}(pv)}{\begin{array}{c}\langle \texttt{return}\ var,\ (C, S, P, I, pf, df, pv)\rangle \\ \to (C, S, P, I, pf, df, pv[ret \leftarrow pv(var)])\end{array}}$$

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

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

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

Figure 3: Symbolic execution of IMP .

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. We also formally present the `interpret` function as Algorithm 1.

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$^{\text{FO}}$ 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 $n \in C \cup P$ **:**
    $\varphi \leftarrow R_{ind}(n)$
    for $T^*(x.dir)$ *occurred in* $R_{ind}(x)$ **:**
      neighbor $\leftarrow pf(n, dir)$
      value $\leftarrow$
      `interpret` $(SH, Orcl, T^*,$ neighbor$)$
      $\varphi \leftarrow \varphi[T^*(n)/\text{value}]$
    for $x.f$ *occurred in* $R_{ind}(x)$ **:**
      intval $\leftarrow df($n, f$)$
      $\varphi \leftarrow \varphi[n.f/\text{intval}]$
    return *the value of* φ *in FOL*
  else:
    return $Orcl(R^*, n)$

def `checkTreeness` (SH, n) **:**
  $reach \leftarrow$ [**false**] $* |C \cup S \cup P|$
  $reach(n) \leftarrow$ **true**
  $flag \leftarrow$ **true**
  while *flag* **:**
    $flag \leftarrow$ **false**
    for $c \in C \cup P, dir \in PF$ **:**
      if $pf(c, dir) \in C \cup P \setminus \{c_{nil}\}$ **:**
        if $reach[pf(c, dir)]$ **:**
          return false
        else:
          $reach[pf(c, dir)]$, $flag \leftarrow$ **true**
  return true

Algorithm 1: Partial interpretation of recursive definitions on symbolic heaps

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

C.4 Reduction for Natural Synthesis

We define normal formulas as follows:

Definition C.3. *A* DRYAD^{FO} *formula is* normal *if it can be characterized using finite number of symbolic heaps, i.e., there exist a finite set of symbolic heaps such that for any concrete heap CH, CH satisfies the formula if and only if CH corresponds to one symbolic heap in that set. Moreover, an* IMP *program is* normal *if all of its* `requires`*,* `ensures` *and* `invariant` *clauses are normal.*

For example, the synthesized loop invariant in the `srtl_inverse` example is normal, as the conjuncts *sorted_l**(`h`), *sorted_lseg**(`h`, `v_1`) and `v_1.next` $=$ `v_2` guarantee that all `loc` variables are list heads. Hence every concrete heap satisfying the loop invariant corresponds to the symbolic heap at the top of Figure 6.

The symbolic validity of an IMP program is defined as:

Definition C.4 (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 Theorem 4.1 reduces the problem of synthesizing a valid program to the task of synthesizing a symbolically valid program:

C.5 Proof of Theorem 4.1

Proof. Consider a filled, normal and symbolically valid IMP program, by contradiction, assume it 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 the program is normal, the initial heap satisfies the Hoare-triple's precondition and can be summarized by a symbolic heap. Then by Lemma C.2, this execution has been summarized by one of the nondeterministic symbolic executions, and the *Orcl* function can simply borrow valid values from the real execution. By Definition C.4, this program violates the symbolic validity. The contradiction concludes the proof. □

D. Encoding the Natural Synthesis Problem

D.1 Arrays encoding the symbolic heap

We declare five arrays to encode the symbolic heap:

```
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 ∧ active[0] ∧ ¬symbolic[0] ∧ ¬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 $\mathtt{B_{HEAP}}$:

$$\texttt{assume} \bigwedge_{i<\mathtt{B_{HEAP}},\ j<\mathtt{DIR\_B}} 0 \le dirs[i][j] \le \mathtt{B_{HEAP}}\ ;$$

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*.

D.2 Array manipulations encoding symbolic semantics

The pointer field mutation statement $u.dir := v$ is simulated as follows:

```
1  void loc_mutation(int u, int dir, int v) {
2    /* implement u.dir := v */
3    assert 0 < u, v < B_LVAR;
4    int source = locvars[u];
5    int dest = locvars[v];
6    assert 0 < source < B_HEAP ∧ 0 ≤ dir < B_DIR;
7    assert active[source] ∧ ¬symbolic[source] ∧ ¬semisym[source];
8    dirs[source][dir] = dest;
9  }
```

The assertions in line 3 and 6 guarantee that the indices do not exceed the array bounds. The assertion in line 7 checks our assumption that the mutated node is a valid concrete node, as we mentioned in Section C.2. Then the solver will make sure the absence of runtime error in executing this statement by these assertions. Finally, line 8 finishes 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 in the main article, 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 (we will explain them later in this section). Then the simulation looks as follows:

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

For each function `f(loc` u_1`, ..., loc` u_m`, int` v_1`, ..., int` v_n`)` with precondition $\varphi_{\mathtt{pre}}$ and postcondition $\varphi_{\mathtt{post}}$, we define a function `call_`f to simulate the effect of calling f:

```
1  void call_f(int u_1, ..., int v_1, ..., int ret,
2    ref int[LVAR] old_locvars, ref int[IVAR] old_intvars,
3    ref int[HEAP] old_symbolic, ref int[HEAP] old_semisym)
4  {
5    assert φ_pre(u_1, ..., v_1, ...);
6    snapshot(old_locvars, old_intvars,
7         old_symbolic, old_semisym);
8    locvars[ret] := havoc(u_1, ...,
9         old_locvars, old_intvars,
10        old_symbolic, old_semisym);
11   assume φ_post(ret, u_1, ..., v_1, ...,
12        old_locvars, old_intvars,
13        old_symbolic, old_semisym);
14 }
```

D.3 Uninterpreted functions encoding nondeterminism

We declare the following uninterpreted functions:

```
bit unint_pf(int n, int dir, int timestamp);
int unint_df(int n, int df, int timestamp);
bit unint_p(int n, int timestamp);
...
int unint_i(int n, int timestamp);
...
```

where p^*/i^* represents a typical recursive predicate/function. Intuitively, $\mathtt{unint\_pf}(n, dir, ts)$ indicates whether the pointer field dir of a concretized symbolic node n is c_{nil} at timestamp ts; $\mathtt{unint\_df}(n, df, ts)$ indicates the value of the data field df of a concretized symbolic node n at timestamp ts. $\mathtt{unint\_p}(n, ts)$ and $\mathtt{unint\_i}(n, ts)$ return the values of recursively defined p^* and i^* at timestamp ts, respectively. All these uninterpreted functions accept integer inputs representing the id's of these nodes, fields and timestamps. The value of the timestamp is bounded by the length of the symbolic execution, assumed a constant $\mathtt{B_{TIME}}$.

The timestamp is maintained as a global variable ts, which starts from 0 at the beginning of a symbolic execution and increments whenever the execution unfolds a node in the symbolic heap. In fact, when ts increments, only the unfolded node in is modified, i.e., those uninterpreted functions on the rest of the heap should remain unchanged. This fact allows us to inquiry the uninterpreted functions only when they are independent from each other. To this end, we introduce a global array

```
bit[BHEAP][BTIME] unchanged;
```

Intuitively, if $unchanged[n][t]$ is true, then all recursive definitions related to n are unchanged when the timestamp increments from $t-1$ to t. The two-dimension array is initially all false; whenever a node n is unfolded and the timestamp increments to t, we convert $unchanged[m][t]$ to true for all nodes m other than n. Then when a nondeterministic value on a node n at timestamp t is needed, we lazily inquire the uninterpreted function only if $unchanged[n][t]$ is false; otherwise keep reducing t until an independent timestamp is reached:

```
int independent_ts(int loc, int ts) {
  while (unchanged[loc][ts]) ts--;
  return ts;
}
```

Now the functions assumed previously for exploring nondeterminism can lazily call corresponding uninterpreted functions with the current ts (after checking index bounds). is_nil and get_int are implemented as:

```
bit is_nil(int loc, int pfld) {
  int real_t = independent_ts(loc, ts);
  return unint_pf(loc, pfld, real_t);
}
int get_int(int loc, int dfld) {
  int real_t = independent_ts(loc, ts);
  return unint_df(loc, dfld, real_t);
}
```

And for each recursive definition R^*, sym_R is implemented as:

```
[ret_type] sym_R(int loc) {
  int real_t = independent_ts(loc, ts);
  return unint_R(loc, real_t);
}
```

Other part of the algorithm can be straightforwardly implemented. Furthermore, we can leverage Algorithm 1 to interpret arbitrary DRYAD$^{\text{FO}}$ formula. 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 $dirs[locvars[v]][dir]$.

D.4 Generator functions encoding holes

We describe the space of possible code fragments that can be used to fill these holes using *generator functions* with the same names. Then a hole `cond(`C`)` can be translated to a call to the following generator function with input C (assuming variable equality check only, written in SKETCH syntax):

```
generator bit cond(int C) {
  for (int 0 ≤ i < C) {
    if (gen_equality()) return false;
  }
  return true;
}
generator bit gen_equality() {
  int bound = {BLVAR | BIVAR};
  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 C 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.

D.5 Proof of Theorem 5.1

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 $encode(\mathtt{sk})$, which does not violate any assertion. Neither does E. □